

AFRL-IF-RS-TR-2003-143
Final Technical Report
June 2003



DEPENDENCE GRAPHS FOR INFORMATION ASSURANCE OF SYSTEMS

GammaTech, Incorporated

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J778


APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

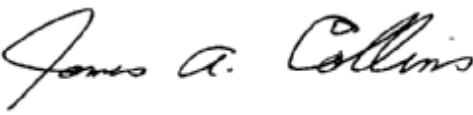
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-143 has been reviewed and is approved for publication.

APPROVED: 
NANCY A. ROBERTS
Project Engineer

FOR THE DIRECTOR: 
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE JUNE 2003	3. REPORT TYPE AND DATES COVERED Final Apr 00 – Jul 02	
4. TITLE AND SUBTITLE DEPENDENCE GRAPHS FOR INFORMATION ASSURANCE OF SYSTEMS			5. FUNDING NUMBERS C - F30602-00-C-0080 PE - 63760E PR - IAST TA - 00 WU - 11	
6. AUTHOR(S) Ray Teitelbaum				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) GrammaTech, Incorporated 317 North Aurora Street Ithaca New York 14850			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/ITB 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-143	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Nancy A. Roberts/ITB/(315) 330-3566				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) Although information flows are critical for understanding assurance and survivability of systems and system designs, tools for understanding information flows in systems are poor. This project sought to provide better tools by exploiting recent advances in tools for understanding information flows in sequential programs using dependence analysis, which provides a sound basis for understanding such information flows. The goal was to develop SystemSurfer, a tool for the information-flow properties of UML designs, and the Information Assurance Workbench, a system for finding assurance problems in programs. These systems were to be based on CodeSurfer, our program-understanding tool. The application of these techniques to UML designs required the design of extensions to the dependence analysis to support concurrency and asynchronous transfer of control. To improve accuracy, it was necessary to consider using abstract interpretation. This project was terminated early because of the cancellation of the IASET project, but not before achieving results in the area of dependence-graph representations, and queries for software assurance. The results of the research are described in the appendices.				
14. SUBJECT TERMS Information Flow, Dependence Graphs, UML, Information Assurance, Slicing			15. NUMBER OF PAGES 72	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1. Summary	1
2. Problem	1
3. Opportunity	1
4. Goals	2
5. Starting Point	2
6. Approach	2
7. Results	3
8. Conclusions	7
9. Appendices	8
Appendix A. A. Binkley, D., Horwitz, S., Macolini, K., Anderson, P., and Teitelbaum, T., Slicing Rose/RT Specifications	9
Appendix B. GrammaTech, A UML-RT Model of the NRL Pump.	31
Appendix C. Kumar, S. and Horwitz, S., Better slicing of programs with jumps and switches.	34
Appendix D. Ezick, J., Richardson, David W., and Teitelbaum, T., Practical Model Checking and Example Generation for Context-Free Processes	49
Appendix E. Anderson, P. and Teitelbaum, T., "Software Inspection Using CodeSurfer.	60

1. Summary

This is the final report on DARPA contract F30602-00-C-0080, “Dependence Graphs for the Information Assurance of Systems”. The project was originally part of DARPA’s Information Assurance Science and Engineering Tools (IASSET) project. Upon the cancellation of IASSET by DARPA, the effort was reassigned to the Organically Assured and Survivable Information Systems (OASIS) project, where it was refocused and wrapped up early.

2. Problem

The motivation for the project was the observation that although information flows are critical for understanding assurance and survivability of systems and system designs, current tools for understanding information flows in systems are poor. *Ad hoc* approaches are unsystematic, error-prone, and lack firm semantic foundation. Formal approaches like code verification are slow, require too much expertise, and do not scale; high-level modeling is too abstract and too remote from actual systems.

3. Opportunity

The project sought to remedy the paucity of tools for analyzing information flows in systems and system designs by exploiting recent advances in tools for understanding information flows in *sequential programs*. Dependence analysis provides a sound and tractable basis for understanding such information flows. The theory of dependence graphs and program slicing, as developed by compiler and software engineering researchers, had matured. There had been considerable investment in generic dependence-graph tools and component technology for sequential code, and products were then emerging. In contrast to the situation for sequential programs, the theory of dependence analysis for *concurrent programs, systems, and system designs*, and corresponding tool support, was lagging behind.

At the same time, UML had become a *de facto* standard for design, and Rational *Rose* had achieved market dominance. Thus, a tool for analyzing information flows based on *Rose* had the potential for significant impact on DOD and industrial practice. By basing our work on *Rose for Real Time (Rose/RT)*, which includes code generation, we planned to address both system designs and system implementation in the same integrated framework.

4. Goals

The specific goal of the project was the development of two prototypes: *SystemSurfer* and the *Information Assurance Workbench*.

SystemSurfer was to be a generic tool for static analysis of dependences in systems and their designs. It would support heterogeneous system descriptions consisting of UML, code, and (as an option that was never supported) even hardware descriptions. It would support both a model-centered view (with automatic modeling of legacy code modules that could be incorporated in system design) and a code-centered view (in which the environment for code could be modeled in UML). *SystemSurfer* was to be both an interactive GUI-based system, and an API for programming customized batch applications.

The *Information Assurance Workbench* (IAW) was to be an IA&S-specific tool layered on *SystemSurfer*. The IAW was to address such sample IA applications as the analysis of covert channels, buffer overruns, and trusted/untrusted separation.

5. Starting Point

The starting point for the project was two existing COTS systems: *CodeSurfer* (from GrammaTech) and *Rose for Real Time* (from Rational).

CodeSurfer is a tool for static analysis of dependences in sequential code, based on a 10-year DARPA project at University of Wisconsin. *CodeSurfer* is both a packaged COTS product and reusable component technology. It has a commercial-strength front end for ANSI C, and experimental front ends for Jovial, Verilog, VHDL, and SPIN. Front ends for C++, x86 machine code, and JVM are under development.

Rose for Real Time (*Rose/RT*) is a tool for system designs and implementations. It is marketed as a COTS product with OLE interfaces. *Rose/RT* supports standard UML extended with ObjecTime's ROOM concepts: capsules; ports; protocols. *Rose/RT* programs are a collection of lightweight threads that communicate through ports with well-defined protocols, modeled by finite-state machines, with transition actions written in C, C++, or Java.

6. Approach

The plan was to build *SystemSurfer* by using and extending *CodeSurfer*'s component technology, and then to build the *Information Assurance Workbench* as a *SystemSurfer* client. The validation plan was to evaluate the prototypes with the aid of GrammaTech customers Sandia National Laboratory, Institute for Defense Analysis (IDA), and the Naval Research Laboratory (NRL), and with other IASET contractors.

The main areas where the CodeSurfer dependence-graph component technology needed to be extended, and where substantial research was needed, were:

- Concurrency (dependence modeling of processes, synchronization, and communication),
- Asynchronous control (dependence modeling of interrupts, exceptions, and aborts)
- Abstract interpretation (symbolic propagation of events, and the decomposition of message-header analysis).

Because SystemSurfer was intended to work on heterogeneous system descriptions that involved both UML designs and code modules, these issues needed to be addressed for both *Rose/RT* models and C/C++ code. Additional areas in the existing CodeSurfer infrastructure where research was needed to support *SystemSurfer* and *IAW* requirements included:

- Precision and performance improvements
- User interface improvements

7. Results

SystemSurfer

The goal of the work was to design a program dependence graph (PDG) representation for concurrent UML state-machine models. Such a dependence-graph representation would permit useful slicing operations to be carried out on *Rose/RT* specifications for various purposes: to help programmers understand an existing specification, to help uncover errors in a specification, to understand the impact of a proposed change to a specification, and to help create new specifications by refining or combining existing specifications.

- The plan we devised appears as Appendix A of this report:

Binkley, D., Horwitz, S., Macolini, K., Anderson, P., and Teitelbaum, T.,
Slicing Rose/RT Specifications, GrammaTech report, December 4, 2002.

- Our work on SystemSurfer was to have been validated, in part, using a UML model of the NRL Pump. The model we developed for that purpose appears as Appendix B of this report:

GrammaTech, *A UML-RT Model of the NRL Pump*.

- Two alternatives were considered for accessing the UML state-machine models.
 - The first possibility was to access the model directly using the *Rose/RT* open OLE interface. The advantage of this approach is that it would be based directly on the UML state machines, and would therefore be independent of the implementation language (C, C++, or Java) into which the diagrams are translated by *Rose*. The disadvantages are (a) it would be problematic to be

sure that we captured the exact semantics of *Rose*'s state machines, and (b) it would have to be maintained as *Rose* evolved.

- The second possibility we considered for obtaining the dependence graphs for UML state machines was to apply CodeSurfer/C to the C code that *Rose* can generate for models. The main advantage of this approach is that it guarantees that we capture the exact semantics of *Rose*'s state machines, as it is based on the C code *Rose* generates for implementation purposes. The main disadvantage is a possible loss of precision caused by the use interpretation and variables in the generated code instead of explicit control flow. In particular, the central section of the generated code is a state-machine interpreter written as a set of nested switch statements.

In analyzing the pros and cons of this approach, we identified show-stopping loss of precision associated with CodeSurfer's treatment of switch-statements. We addressed this problem, and solved it in the publication:

Kumar, S. and Horwitz, S., Better slicing of programs with jumps and switches. In *Proceedings of FASE 2002: Fundamental Approaches to Software Engineering*, (Grenoble, France, April 8-12, 2002).

The paper is attached as Appendix C of this report. The solution has been implemented and transitioned into the COTS *CodeSurfer* product.

When the project was reassigned from IASET to OASIS, the focus shifted and further work on SystemSurfer was discontinued.

Information Assurance Workbench

SystemSurfer would have been a general-purpose tool for understanding systems; in contrast, the role of the *Information Assurance Workbench (IAW)* was to support operations more narrowly focused on information assurance needs. The *IAW* would have been a kind of laboratory for developing and applying new kinds of information-assurance analyses enabled by the availability of the system-wide dependence information provided by *SystemSurfer*. We aimed to support three specific analyses: covert channel identification, trusted subset identification, and trusted/untrusted separation.

Detailed work in support of our aims included the following:

- **New ways to pose dependence queries.** In discussion with security analysts at Sandia National Labs, we learned that in order to support their preferred way to pose information-flow analysis questions, it was necessary to support queries posed in terms of variables, not program points. Accordingly, we introduced new query-input-modes for the dependence queries. The mode can be one of the following:
 - In "point mode", a query can be posed in terms of a set of points in the program, but the variables that are used or defined at the points are not distinguished.

- In “variable mode”, the user can pose queries in terms of the variables in the program. For example, a variable forward slice takes as a parameter a set of variables. It first finds all points in the program where the variables are defined, and then does a regular forward slice from that point.
- In “point-and-variable mode,” the user can pose queries in terms of individual variables that are used or defined at a particular set of points in the program. This can also be thought of as queries in terms of edges in the dependence graph. Any point in the program may either use or define a set of variables. This mode allows the user to select a subset of the variables from which to start the query.
- In “function mode”, the user can pose queries in terms of the names of functions in the program.

All of the basic queries—slice, backward slice, chop, truncated chop, predecessors and successors—were extended to allow queries to be posed in these terms.

- **New queries to reveal program flows.** We implemented a prototype model checker to support path queries posed in computation-tree logic (CTL). The goal of the work was to provide an open and expressive language for code-based assurance queries in the Information Assurance Workbench (IAW). A description of the prototype we developed as Appendix D of this report:

Ezick, J., Richardson, David W., and Teitelbaum, T., *Practical Model Checking and Example Generation for Context-Free Processes*, Technical Report, May 15, 2001.

This work has been continued under the auspices of a DARPA SBIR Phase II contract: “Verification of Hierarchical Graph Structures.

- **Performance and precision improvements in the CodeSurfer infrastructure.**
 - *Pointer analysis.* The precision of the dependence graph and the scalability of the entire system are highly dependent on the accuracy of the pointer analysis algorithms used. Higher accuracy yields fewer false-positive results. Also, the more accurate the pointer analysis, the more scalable are the algorithms for building the dependence graph.

We were able to achieve significant speed increases. In addition to making speed improvements, we extended the algorithm to support specialized treatment of structure fields. Our effort on pointer analysis broke the back of this problem in the sense that it is no longer the bottleneck it once was.

- *Context sensitive GMOD/GREF.* We improved the precision of the analysis of global variables usage at call sites. The previous situation can be described in terms of the following example:

```
int x,y;
void f(*int p) { *p = 0; }
```

```

void m(void) { f(&x); }
void n(void) { f(&y); }
void main(void){ m(); n(); }

```

In this example, the variables x and y *both* showed up as being modified by *both* $m()$ and $n()$. This is the GMOD set. However, it is evident from the code that m can only modify x , and n can only modify y . This situation shows up in many examples. It was particularly apparent when running the system on the `sendmail` benchmark.

The solution is to have a pass in the analysis that filters the GMOD set to exclude variables that cannot possibly be accessed by a given call site.

The following table shows the effect of the optimization on some of our standard benchmark programs. Each column has three entries. The first shows the metric before the optimization was implemented. The second entry is after the implementation. The third is the percentage change.

		Size of SDG (Mb)			Summary edge time			Total build time			Forward slice time		
Program	LOC	before	after	change	before	after	change	before	after	change	before	after	change
compress	1937	1.15	0.59	-48%	0.05	0.05	0%	1.98	0.75	-62%	0.02	0.01	-50%
cpp	4079	4.35	3.66	-16%	1.4	0.94	-33%	11.31	9.16	-19%	0.36	0.19	-47%
byacc	6626	4.53	4.64	3%	0.26	0.32	-23%	9.09	6.66	-27%	0.44	0.29	-34%
cadp	12787	9.36	7.42	-21%	0.73	0.83	14%	21.58	13.15	-39%	0.22	0.15	-32%
flex	12400	6.47	6.09	-6%	1.89	1.97	4%	19.21	13.91	-28%	0.87	0.58	-33%
ijpeg	28177	19.76	15.02	-24%	7.59	1.44	-81%	532.66	563.75	6%	4.67	1.37	-71%
go	29246	24.45	19.21	-21%	10.47	7.2	-31%	59.87	39.21	-35%	12.81	6.86	-46%
ntpd	61068	40.24	35.44	-12%	41.73	34.99	-16%	281.25	205.36	-27%	22.75	20.11	-12%

- **Outreach.** We participated in the Workshop on Inspection in Software Engineering in Paris, 2001, where we explained the relevance of our DARPA-supported work to software inspection. The workshop paper appears as Appendix E of this report:

Anderson, P. and Teitelbaum, T., “Software Inspection Using CodeSurfer.”
Workshop on Inspection in Software Engineering (CAV 2001), (Paris, France.,
July 18-23, 2001).

Two journal papers based on this report are currently under review for publication in IEEE Software, and IEEE Transactions on Software Engineering.

8. Conclusions

We believe that our work was on track and making good progress at the time DARPA cancelled its IASET project. Better tools for understanding information flows in systems and system designs are still needed, and we believe the opportunity to address that need in the manner we proposed still exists.

9. Appendices

- A. A. Binkley, D., Horwitz, S., Macolini, K., Anderson, P., and Teitelbaum, T., *Slicing Rose/RT Specifications*, GrammaTech report, December 4, 2002.
- B. GrammaTech, A UML-RT Model of the NRL Pump.
- C. Kumar, S. and Horwitz, S., Better slicing of programs with jumps and switches. In Proceedings of FASE 2002: Fundamental Approaches to Software Engineering, (Grenoble, France, April 8-12, 2002).
- D. Ezick, J., Richardson, David W., and Teitelbaum, T., Practical Model Checking and Example Generation for Context-Free Processes, Technical Report, May 15, 2001
- E. Anderson, P. and Teitelbaum, T., “Software Inspection Using CodeSurfer.” *Workshop on Inspection in Software Engineering (CAV 2001)*, (Paris, France., July 18-23, 2001).

Slicing Rose/RT Specifications *

David Binkley, Susan Horwitz, Kirk Macolini, Paul Anderson, Tim Teitelbaum
GrammaTech Inc.
317 N. Aurora St., Ithaca, NY 14850

December 4, 2000

1 Introduction

This report summarizes GrammaTech's work on the design of program dependence graphs (PDGs) to represent Rose/RT specifications. The goal of the work was to design a PDG that would permit useful slicing operations to be carried out on the Rose/RT specification; for example, to help programmers understand an existing specification, to help uncover errors in a specification, to understand the impact of a proposed change to a specification, and to help create new specifications by refining or combining existing specifications.

The first part of the report provides background material on the contents of a Rose/RT specification; the second part describes how to build a PDG to represent a Rose/RT specification; the third part gives some examples; and the fourth part discusses some open questions.

2 Background on Rose/RT

There are three major kinds of diagrams used in a Rose/RT specification: the *class diagram*, the *state diagram*, and the *structure diagram*. The contents of these diagrams are discussed below, using the example in Figures 1, 2, 3, 4, and 5. (See "Using UML for Modeling Complex Real-Time Systems" for more detailed definitions.)

2.1 Class Diagram

The class diagram shows relationships between two stereotypes of a UML class: *capsules* and *protocols*. It shows aggregation, but does not show any connections; those are given in the structure diagrams. The example class diagram shown in Figure 1 defines five classes: *terminal*, *call*, *HardwareInterface*, *Voice*, and *TerminalHook*. The icons in the upper-right corners indicate that *terminal*, *call*, and *HardwareInterface* are capsules, and that *Voice* and *TerminalHook* are protocols. More information about the contents of a class diagram is provided below.

Capsules: A *capsule* describes the structure of an entity in the specification. Each instance of a capsule is a lightweight thread whose behavior is described by the finite-state machine that is specified in the capsule's state diagram. Each capsule is drawn as a box divided into four parts (by horizontal lines). The contents of those parts are class-name, attributes, operations, and ports. Example: The *call* capsule has class-name "call", has no attributes, has one operation "class.partner()", and has one port "terminalRole".

*This work was funded by DARPA contract F30602-00-C-0080 (under project "Dependence Graphs for Information Assurance of Systems" in program "Information Assurance Science & Engineering Tools" (IASSET))

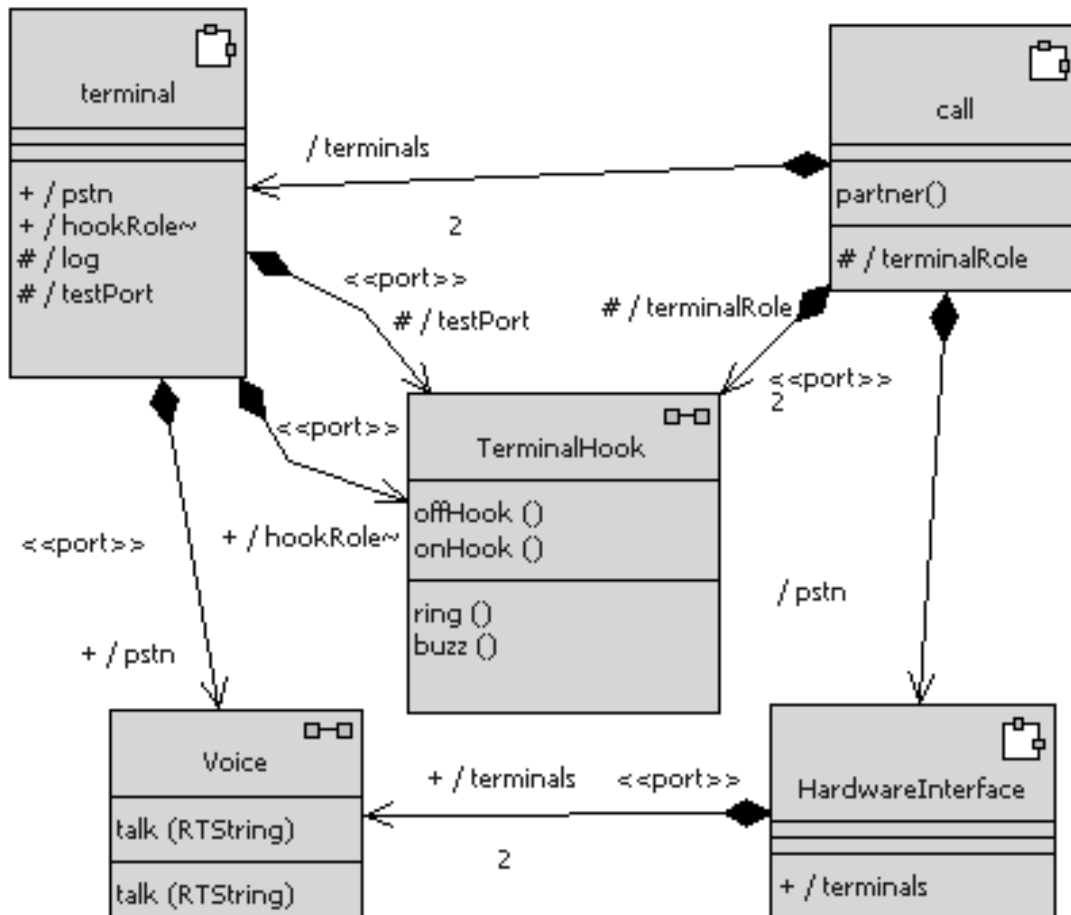


Figure 1: Example Class Diagram.

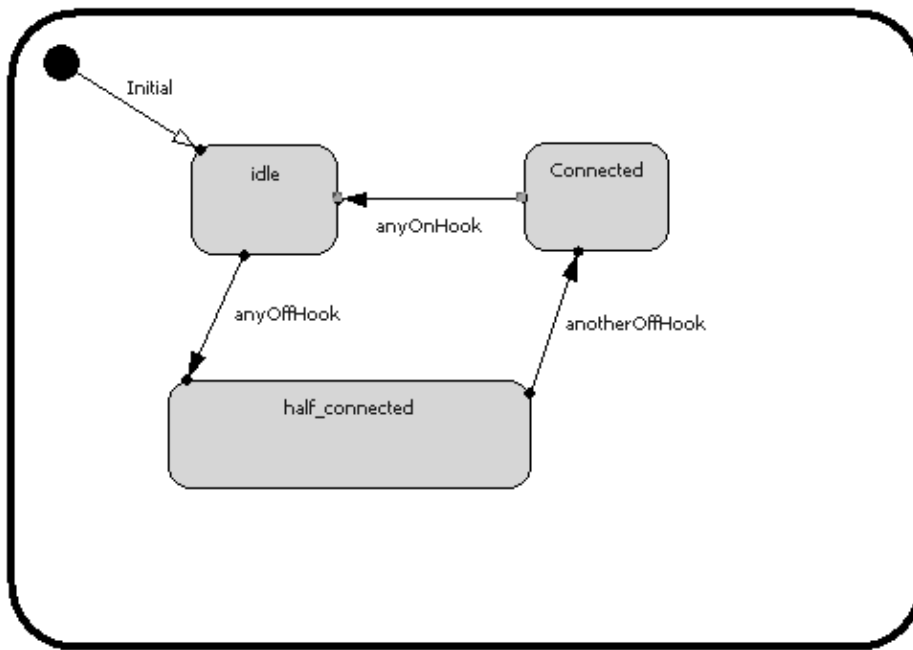


Figure 2: Example State Diagram (for the Call Capsule).

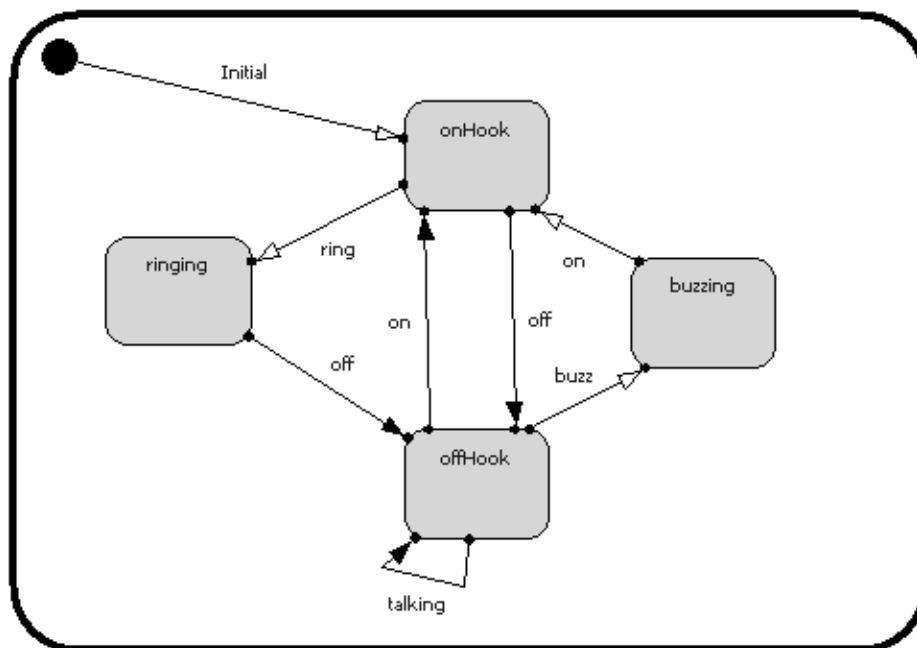


Figure 3: Example State Diagram (for the Terminal Capsule).

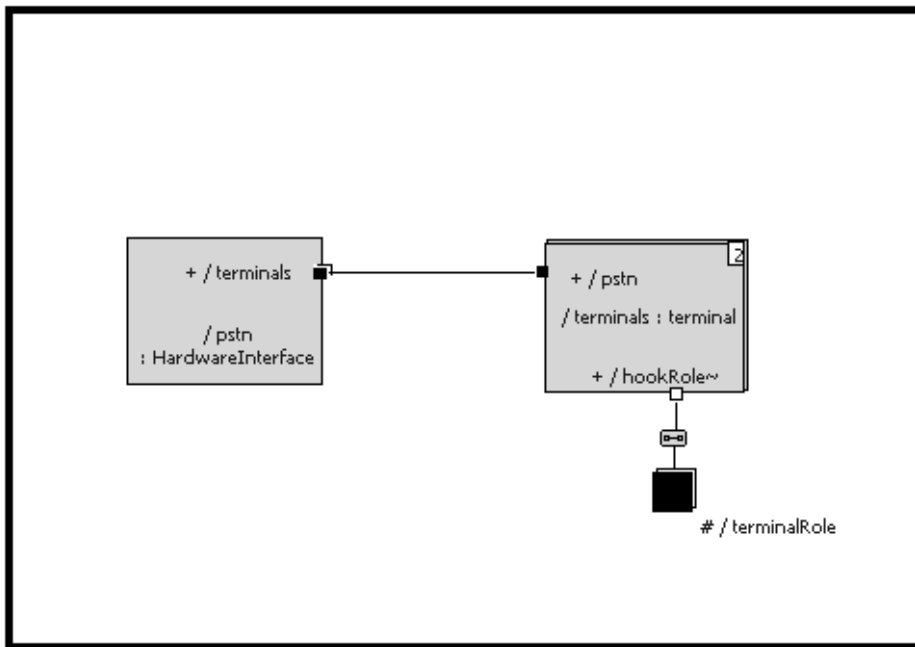


Figure 4: Example Structure Diagram (for the Call Capsule).

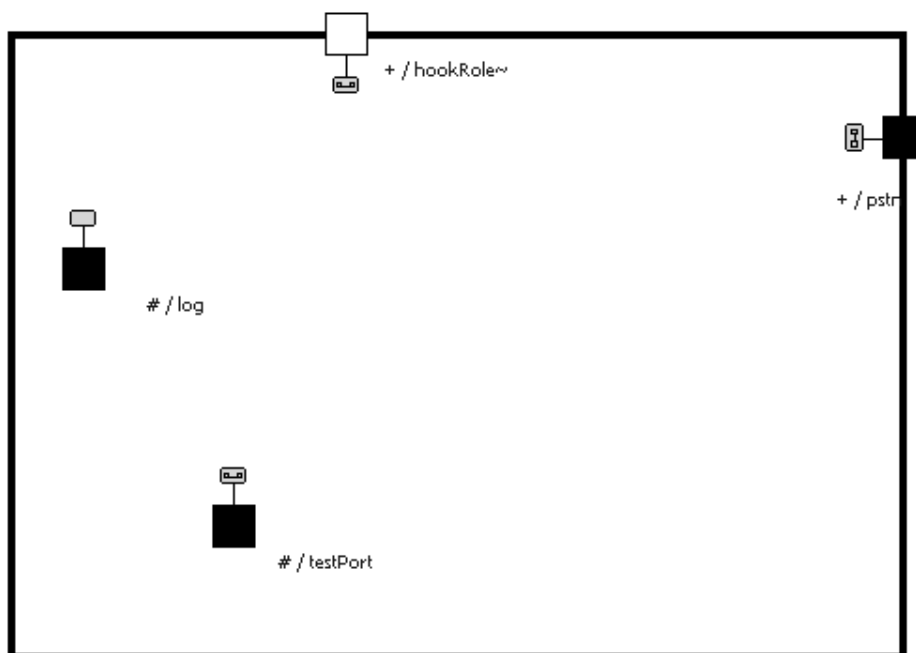


Figure 5: Example Structure Diagram (for the Terminal Capsule).

Protocols: A *protocol* describes the language of valid signals. It is possible to specify that this language must be the language of a given finite-state machine; however, this feature is not in common usage, nor is it enforced by Rose/RT. Each protocol box is divided into three parts (by horizontal lines). The contents of those parts are protocol-name, incoming signals, and outgoing signals. There are four special built-in protocols: Log, Timing, Frame and Exception. Example: The *TerminalHook* protocol has protocol-name “TerminalHook”, two incoming signals “offHook()” and “onHook”, and two outgoing signals “ring()” and “buzz”.

Associations: Edges connecting classes indicate associations between capsules and protocols. The edges between capsules with a black diamond at one end and an arrow head at the other end mean “sub-capsule” (however, note that a sub-capsule is not a sub-class). For example, the *call* capsule has two sub-capsules: *terminal* and *HardwareInterface*. A capsule and its sub-capsules “live and die” together. For example, a *HardwareInterface* (sub)-capsule is created when a *call* capsule is created, and is destroyed when the *call* is destroyed.

The directions of the edges indicate “navigability.” For example, a *call* can tell you what its associated *HardwareInterface* is, but not vice versa. The edges from a capsule to a protocol mean that one or more of the capsule’s ports uses that protocol. The edge is labeled with the port name and with “<<port>>.”

Role Names: Edges between capsules (indicating a sub-capsule relationship) can be labeled with “role names” (e.g., the edge from *call* to *terminal* is labeled “terminals”). These role names are used in structure diagrams. For example, in the structure diagram for *call* (shown in Figure 4), one box has the text “/pstn” and the other has “/terminals.” Those two names are the role names for the sub-capsules of a *call*. If a capsule has two instances of a sub-capsule that play different roles (e.g., are wired up to other sub-capsules in different ways) then the role name used in the class diagram would clarify which sub-capsule instance in the structure diagram was playing which role.

Multiplicity: Numbers on edges indicate “multiplicity.” For example, a *call* has **two** *terminal* sub-capsules and a *HardwareInterface* has **two** ports named “terminals” that use the *Voice* protocol.

2.2 State Diagram

2.2.1 States and Transitions

A capsule’s state diagram defines the behavior of all instances of that capsule. The behavior specified used a finite-state machine (FSM). As usual, this FSM includes a set of states (one of which is a special initial state), and a set of transitions from state to state. Each transition is labeled with a *trigger* of the form $s_1, s_2, \dots, s_n : p_1, p_2, \dots, p_k$, where each s_i is a signal and each p_j is a port (the special symbol * can also be used to mean *any* signal). The idea is that the transition is to be taken if any of the signals s_i is received on any of the ports p_j . Note that to be a valid specification, all of the signals must be available on all of the ports; i.e., in the class diagram, for each signal s_i and each port p_j :

1. port p_j must be specified for this capsule, and
2. there must be an association between this capsule and a protocol that includes signal s_i .

A transition may also have a guard (consisting of a C++ boolean expression), in which case the transition is only taken if the guard is true. In addition, a transition can have one or more *choice points*, each with an associated boolean expression. Choice points allow the transition to be made to *different* successor states depending on the values of the boolean expressions: at each choice point, the value of the expression determines which of the two succeeding segments of the transition is taken. Note that the use of a choice point with boolean expression B is not the same as having

two guarded transitions, one guarded with B and the other guarded with $!B$. This is because when a guard is false, not only is the transition not taken, but the trigger is “consumed” (i.e., the FSM must wait until another signal is received before attempting to take a transition).

States and transitions can be labeled with actions; a state can include both an *entry* action (to be performed each time the state is entered) and an *exit* action (to be performed each time the state is exited). Transition actions are performed each time the transition is taken. Actions are arbitrary C++ code, possibly including *message sends*. A message consists of a signal and optionally other data, sent on a particular port. Assuming that there are no dynamic instances of capsules or rewiring of ports, the destination of the message is determined by the wiring together of ports specified in the structure diagram (see Section 2.3 below).

In a Rose specification, an edge from state to state with a black arrowhead means that there is an action associated with that transition. A white arrowhead means there is no action. In the display of a Rose/RT state diagram – and thus in the example state diagrams given here – triggers, guards, and actions are all represented using identifiers; the actual trigger, guard, or action code can be viewed through the use of a dialog box.

2.2.2 Hierarchical FSMs and History

A capsule’s FSM can be *hierarchical*: a state can be defined in terms of a sub-FSM (i.e., a set of substates). A sub-FSM can be entered in three ways:

1. entry to the initial state,
2. entry to a specific substate, or
3. entry via history (described below).

Each state of a sub-FSM inherits the outgoing transitions of its parent state. For example, suppose that FSM S consists of a sub-FSM that includes a state s . If s has no outgoing transition labeled X , but S does have such an outgoing transition to state T , then from (sub)-state s , the transition to T is taken on input X .

As mentioned above, a transition to a state S with substates can be done via *history*. In this case when entering S the particular substate entered is the last substate visited on the previous visit to S . If S has never been visited before, then the substate that is entered via history is its FSM’s initial state.

Example: In the state diagram in Figure 9, state M has three substates: an initial state, state *Wash* and state *Dry*. When state M is entered via the transition labelled “init” from the top-level FSM’s initial state, it is the initial state of the sub-FSM that is entered (because the specification includes no explicit transition into the sub-FSM on “init”). When state M is entered via the transition labeled “done” from state *Handler*, M ’s sub-FSM is entered via history (because the specification includes an edge labeled “done” that enters the sub-FSM from outside, and whose target is the special history state, denoted using a small circle labeled “H”). If the sub-FSM had most recently been in state *Wash* when it received an “interrupt” signal, then the re-entry to the sub-FSM via history would cause state *Wash* to be re-entered (and similarly, if it had most recently been in state *Dry* when it was interrupted, it would be the *Dry* state that would be re-entered via history).

2.2.3 The Initial State

The initial state of a FSM is indicated in a state diagram by a black circle. The transition out of the initial state is labeled with a special “initial event”. Conceptually, the initial event is sent to all capsules by the system when the program starts. No other FSM transitions can be taken until all FSMs have finished taking their initial-event transitions. In the case of sub-FSMs, the initial transition is taken under two circumstances:

- when the sub-FSM is entered via entry to its initial state, or
- when the sub-FSM is entered via history, but the sub-FSM has not been previously visited.

2.3 Structure Diagram

A capsule’s *structure diagram* is used to specify the patterns of communication between that capsule and its sub-capsules, as well as defining which ports are used to communicate with “the outside world”. The structure diagram shows all of the capsule’s ports, and shows how its subcapsules are wired together. It also specifies the capsule instances and the static connectors used to connect specific port instances.

In a structure diagram, if multiple subcapsules have identical wiring, they can be shown using essentially one rectangle with a multiplier number in the upper right corner (and shadowing to suggest overlaying). For example, in the *call* structure diagram in Figure 4, there are two *terminal* sub-capsules, shown using this technique.

A capsule’s ports are shown as black rectangles with little protocol icons above them (they look like Martians). Ports are the connection points for capsules into which protocols are plugged. There are two kinds of ports that are used in three different ways.

Kind 1: Relay ports Relay ports allow the direct (zero overhead) delegation of signals destined for a capsule to a sub-capsule. Relay ports can only appear on the boundary of a capsule and, consequently, always have public visibility.

Kind 2: End Ports End ports are the ultimate sources and sinks of all signals sent by capsules. To send a signal, a state machine invokes a send or call operation on one of its end ports. End ports may appear on the boundary of a capsule with public visibility. These ports are called public end ports.

Use 1: A port drawn inside the capsule’s rectangle with no connection to a sub-capsule is connected to the outside world and there should be sends and/or receives for this port on some of the capsule’s FSM edges that allow this capsule to communicate with the outside world. In Figure 5, there are two such ports (*log* and *testPort*), which are end ports.

Use 2: A port drawn inside the capsule’s rectangle with a connection to a sub-capsule is *not* connected to the outside world. There should be sends/receives in the capsule’s FSM and in the sub-capsule’s FSM machine allowing the capsule and the sub-capsule to communicate with each other. In Figure 4 there is one such port (*terminalRole*), which is an end port.

Use 3: A port drawn on the edge of the rectangle is connected to some other capsule’s port (not to the outside world). In Figure 5, there are two such ports (*pstn* and *hookRole*). Note that, as specified in the *call* capsule’s Structure Diagram, the *pstn* port is connected to the *HardwareInterface* sub-capsule’s port, and the *hookRole* port is connected to the *call* capsule’s *terminalRole* port. In general, these ports are relay ports if connected to a subcapsule and end ports otherwise.

3 Building a PDG from a Rose/RT Specification

For each instance of a capsule in a Rose/RT specification, there is one CFG (and one PDG) that represents the capsule’s FSM. Edges between the PDGs, called *message dependence edges* are used to represent the dependences induced by sends and the corresponding receives. (These message dependences are similar to the *interference dependences* that have been defined in the literature on program-dependence graphs for concurrent programs [4, 9, 5]. Other than the addition of message dependences (which requires analyzing the Rose/RT specification to find the send/receive patterns), the correspondence between the CFG and the PDG is standard (i.e., once the correspondence between the Rose/RT specification and the CFG has been defined, standard techniques can be used to build the corresponding PDGs).

Building the CFGs for a Rose/RT specification is discussed below, assuming that there are no dynamic instances of capsules or rewiring of ports, and that all messages are asynchronous.

3.1 Building a CFG from a Rose/RT Specification

The correspondence between a capsule's FSM and the nodes of the CFG that represents that FSM are as follows:

- For FSM M , the CFG includes a node labeled “Enter M ” and a node labeled “Exit M ” (thus, the CFG for a FSM is somewhat analogous to the CFG for a procedure).
- Each action in the FSM (associated with an transition, with state entry, or with state exit) is represented by a separate procedure (and thus a separate CFG); instances of the actions are represented in the CFG for the FSM as calls to those procedures (this avoids replication of action code while allowing inlining to remove calls to small or seldom used actions).
- Each *landable* state S produces a fragment in the CFG that begins with a node labeled “Start S ” and ends with a node labeled “End S ” (Figure 7 shows four such fragments). State S is landable if the FSM can come to rest in S . This occurs when S is a simple state (one with no substates) or when S is a composite state (one with nested states) that has no initial transition and there is a transition that ends at S . In the latter case, a transition that ends at S will not continue to any of S 's substates. For example, in Figure 6 states *Off*, *Maint*, *Running*, and *Done* are landable while state *On* is not.
- The nodes of the CFG fragment built for landable state S represent the receiving and processing of messages. The node labeled “Start S ” is followed by a receive node, then a collection of nodes that represent a switch. The switch has one branch for each possible transition out of state S . Each branch is composed of a sequence of calls to the action routines that represent the actions the FSM executes when it takes the corresponding transition. For example, the FSM in Figure 6 includes a transition from state *Off* to state *Maint* with state-exit action A3, transition action A4, and state-entry action A5; thus, in the switch in the sub-CFG that represents state *Off* (shown in Figure 7), there is a branch for **case** *Maint* with the corresponding sequence of calls (shown all in one node to save space in the diagram). The sequence of calls is followed by a goto node whose target is either the state that is the target of the FSM transition, or a *history choice node* (described below). The switch also includes a default case that consumes (and thus ignores) any trigger for which the state has no defined transition. The default case ends with a goto node back to the receive node.
- There are four additional FSM constructs that must be represented in the CFG: transition guards, choice points, inner internal self-transitions, and history.
 1. **Transition guards:** When a transition includes a guard, the corresponding branch of the switch statement in the CFG includes a call to the guard code, which returns a boolean value. That value is used in an **if** node whose true successor is the sequence of action calls described above, and whose false successor is a goto node whose target is the receive node just before the switch; i.e., if the guard is false, then the transition is not taken but the incoming message is consumed, and the FSM waits for another message. Example: The FSM in Figure 6 includes a guard when leaving state *On*. Therefore, its CFG, shown in Figure 7, includes a node labeled “if guard1.” This node actually appears twice in the figure because both substates of *On* must deal with the guard.
 2. **Choice points:** A choice point is also represented by an **if** node. In this case, the true branch includes calls to the action routines that correspond to the actions performed if the condition at the choice point is true, while the false branch includes calls to the actions performed if the condition is false.
 3. **Inner internal self-transitions:** In a Rose/RT specification, a FSM state that has substates can have a special kind of self-loop called an *inner internal self-transition*. When this transition is taken, the only action that is performed is the action associated with that transition (no state-exit or state-entry actions are taken). Therefore, the CFG fragment

that corresponds to one of these transitions includes only a call to the action associated with the transition (there are no calls to state-entry or state-exit actions).

4. **History:** Handling history requires three additional components in the CFG. First, for a state S that contains a transition to history, a new *history-choice* node labeled “ S -HC” is added to the CFG. This node has an outgoing edge for each substate that can be reached through history. For the transition through history to substate T , the history choice node is followed by a series of calls to the entry actions of each nested state encountered between S and T . The final node in this series has an edge to T ’s start state. Example: The FSM in Figure 9 uses history. As a result, its CFG, shown in Figure 10, includes the history choice node labeled “ M -HC.”

The second addition for history involves labels on certain CFG edges. In particular, as noted above, from each history-choice node there is a path in the CFG to the Start node for each substate that can be reached through history; the last edge in each such path is labeled with the name of the reached state. Example: the three edges out of the “ M -HC” node in Figure 10 are labeled *wash*, *dry*, and *M-Init* (the names on their respective target Start nodes).

Also, the edges from goto nodes that represent transitions from within S and its substates to states outside of S , are labeled with the *compliment* of the state being departed (the state name with a bar above it). Example: In Figure 10, these labels are “ \overline{wash} ” and “ \overline{dry} ”.

The motivation for these edge labels is that the presence of history makes certain CFG paths non-executable. For example, if the FSM in Figure 9 exits substate *Wash* of state M and then returns to state M via history, it must go to substate *Wash* (not *Dry*). Thus the CFG path: *Start Wash* \rightarrow *receive* \rightarrow *switch* \rightarrow *case interrupt* \rightarrow *call A3* \rightarrow *goto Start Handler* \rightarrow *Start Handler* \rightarrow *receive* \rightarrow *switch* \rightarrow *case done* \rightarrow *goto M-HC* \rightarrow *M-HC* \rightarrow *Start Wash* is an executable path, while the CFG path that is the same except that it ends with *M-HC* \rightarrow *Start Dry* is *not* executable. In terms of the CFG edge labels, the executable path will include an edge labeled \overline{Wash} followed later by an edge labeled *Wash* (i.e., two *matching* edges), while the non-executable path will include an edge labeled \overline{Wash} followed later by an edge labeled *dry* (i.e., two *non-matching* edges). Thus, executable CFG paths are those whose sequence of labels is in some context-free language (in this case, the language of matching barred and unbarred names), while the sequence of edges labels on non-executable paths are not in that language.

The third addition is a CFG fragment that represents a transition to the initial state of a sub-FSM via history. If a transition ends on history in state S and S has never been visited before then history takes S to its initial state. This is represented in the CFG by a node labeled “Start S -init”, which is the target of an edge from S -HC labeled “ S -init.” There is also a node labeled “End S -init” and a sequence of calls to the action routines that reflect the actions the FSM executes upon a transition to S ’s initial state. Example: In Figure 9 there are no actions associated with the initial transition in M ’s sub-FSM; thus, the resulting CFG fragment (Figure 10) includes only the nodes labeled “Start M -init,” “End M -init,” and “goto Start Wash”. This final node reflects M ’s sub-FSM’s initial transition to state *Wash*.

4 Examples

This section presents some example Rose/RT state diagrams, and the corresponding CFGs and PDGs. In each case, the motivation for considering the example is discussed.

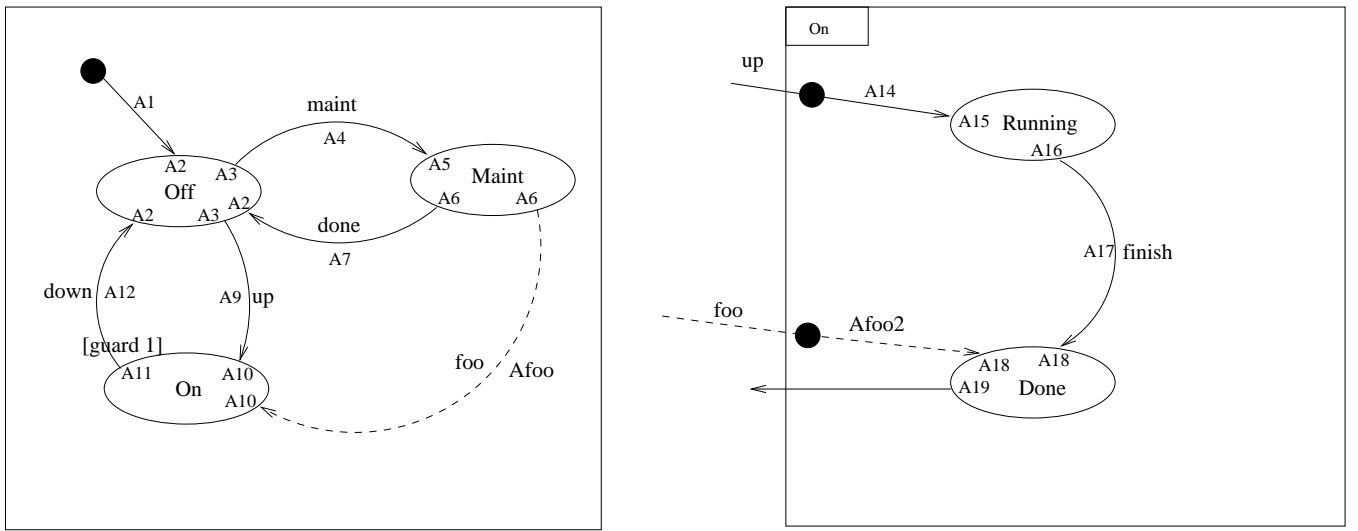


Figure 6: State Diagram for the Hierarchical FSM Example

4.1 An Example with Substates

Figures 6, 7, and 8 show the FSM, CFG, and PDG for an example that illustrates a hierarchical FSM. (In Figure 7, sequences of calls are represented together in a single node to save space. As can be seen in Figure 8, each call is actually represented by a separate node.) The top level FSM has three states *Off*, *On*, and *Maint*. The state *On* has two substates: *Running* and *Done*. This example illustrates two things: First, the effect of a parent state’s outgoing transition on its substates, and second how a guard on such a transition is handled.

As discussed in Section 2.2.2, each state of a sub-FSM “inherits” the outgoing transitions of its parent state. In this example, neither state *Running* nor state *Done* has an outgoing transition labeled “down”, but their parent state, state *On*, does have such a transition (to state *Off*). Note that the transition has a guard (*guard1*). Therefore, when the FSM is in one of the substates, and receives a “down” trigger, the following happens:

- The guard is evaluated; if it is false; nothing further happens. If it is true:
 1. The exit action for the current substate is performed.
 2. The exit action for the parent state (*On*) is performed.
 3. The transition action is performed.
 4. The entry action for state *Off* is performed.
 5. The new current state is State *Off*.

These actions are reflected in the two sub-CFGs for states *Running* and *Done*.

4.2 An Example with History

This example illustrates history. Figure 9 shows the FSM with two states (*M* and *Handler*). State *M* has two substates (*Wash* and *Dry*). The FSM cycles between *Wash* and *Dry* until an interrupt occurs, which causes a transition to *Handler*. When *Handler* finishes it transitions, using history, back to whichever substate of *M* was interrupted.

Figures 10, and 11 show the corresponding CFG, and PDG. First, look at the parts of the CFG that correspond to the FSM transition from state *M* to state *Handler* (due to the receipt of an “interrupt” message): these are the “interrupt” cases in the switches under “Start Wash”

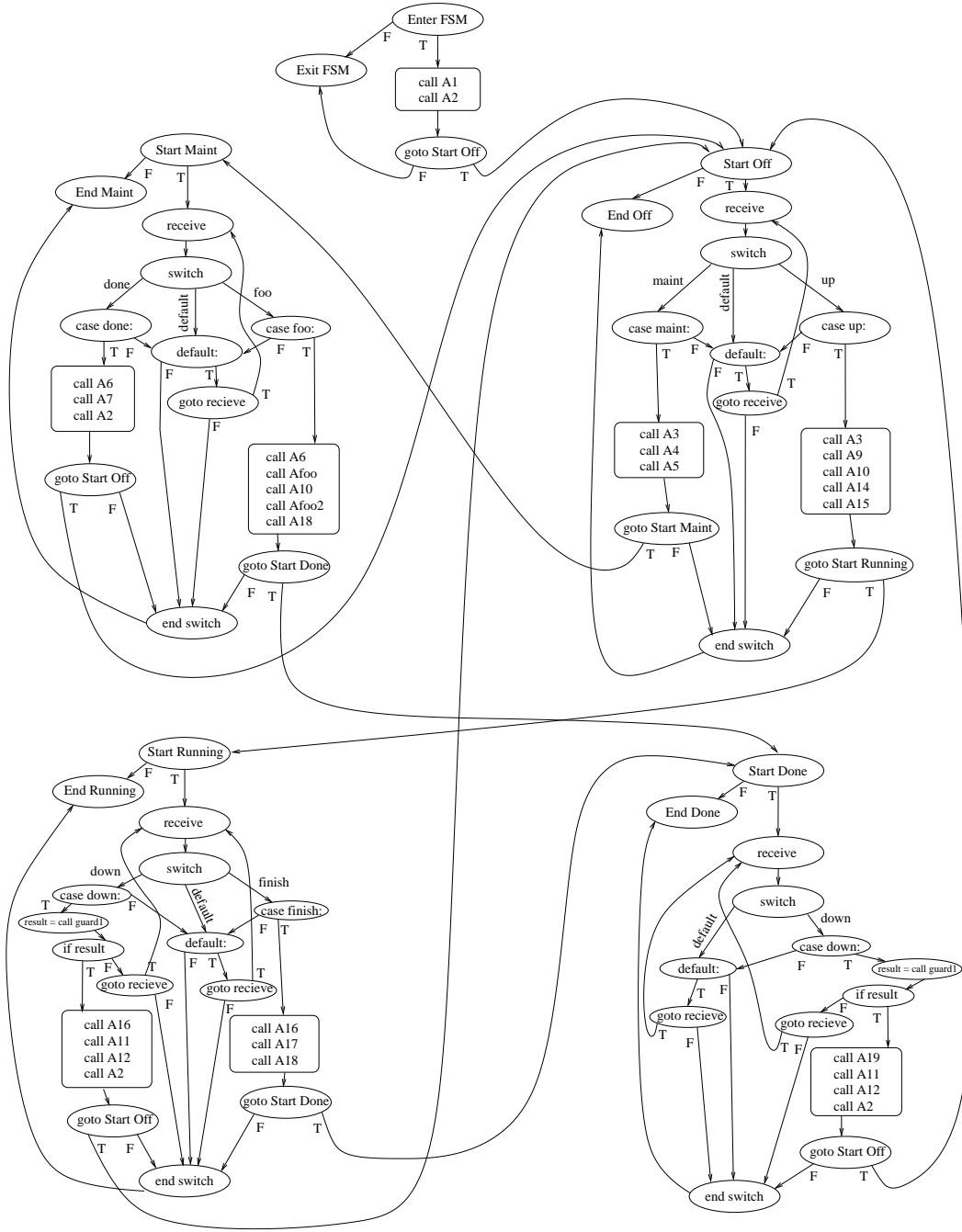


Figure 7: CFG for the Hierarchical FSM Example

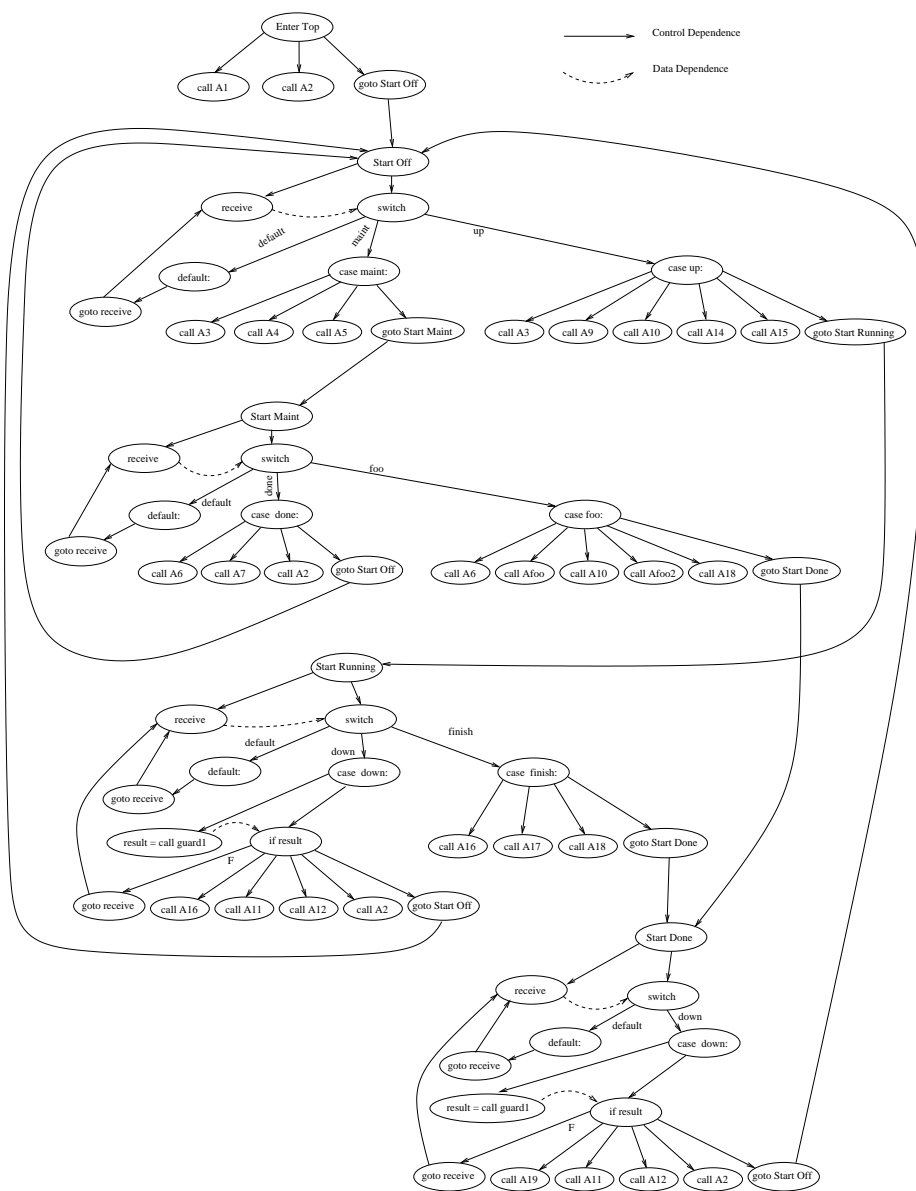


Figure 8: PDG for the Hierarchical FSM Example

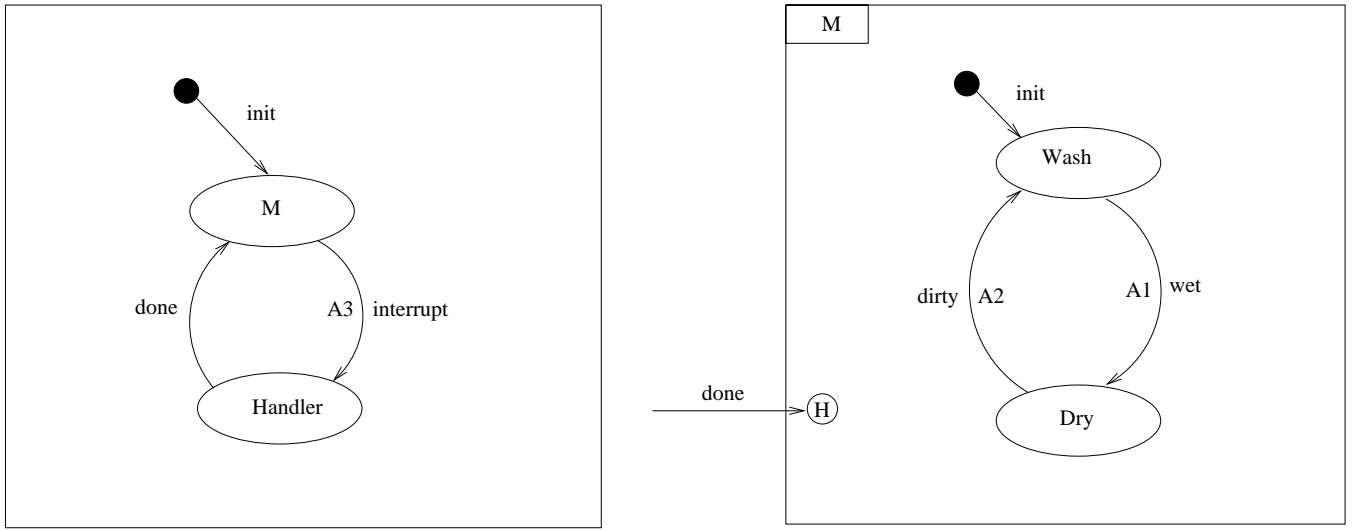


Figure 9: State Diagram for the History Example

and “Start Dry” (since *Wash* and *Dry* are the substates of *M*). The edges from the nodes labeled “goto Handler” to “Start Handler” are labeled “*wash*” and “*dry*” (i.e., the labels correspond to the substates of state *M* that are being exited).

Now look at the part of the CFG that corresponds to the receipt of a “done” message while in state *Handler*. The target of the goto node in that part of the CFG is the history-choice node (labeled “*M*-HC”). The edges out of the history choice node to “Start Wash” and “Start Dry” are labeled *wash* and *dry*. As explained in Section 3.1, these edges are included so that executable CFG paths can be defined as those paths whose sequences of labels are members of some context-free language. The history choice node and the associated edges labels also occur in the PDG, where they serve a similar purpose.

The history choice node also has an outgoing edge labeled “M-init”. This is because, in general, if state *S* includes a transition to history and no substate of *S* has been visited (i.e., on the first visit to *S*), then history takes the machine to the initial state. This state is represented by the sub-CFG that includes the three nodes labeled “M-init”, “goto Start Wash”, and “End M-init”. In this particular example, the machine can never go to *M*’s history without having previously been in state *M* (so the sub-CFG that represents this situation is not necessary). It is possible that static analysis could be used to recognize such situations and thereby omit the unnecessary CFG (and PDG) nodes and edges.

4.3 An Example with Actions and Useful Slices

This example, shown in Figures 12, 13, 15, 14, and 16, illustrates multiple processes (FSMs), and the potential for static analysis to reduce the number of message-dependence edges in the PDG. It also provides an example of slicing a Rose/RT specification. The example involves reading students’ homework and exam scores, and computing each student’s final grade as well as the average homework and score scores for the class as a whole. It includes four processes (four capsules): one to do the reading, one to compute each student’s final grade, one to compute the class’s homework average, and one to compute the class’s exam average.

The transition actions in the state diagrams for this example are important, so they are shown explicitly. For the initial transition in the state diagram for the *reader* capsule, the action is named “**A_main**” and the corresponding code is shown to the right of the diagram. For the other state diagrams, the actions are given on the transitions themselves, using C++ code inside curly braces.

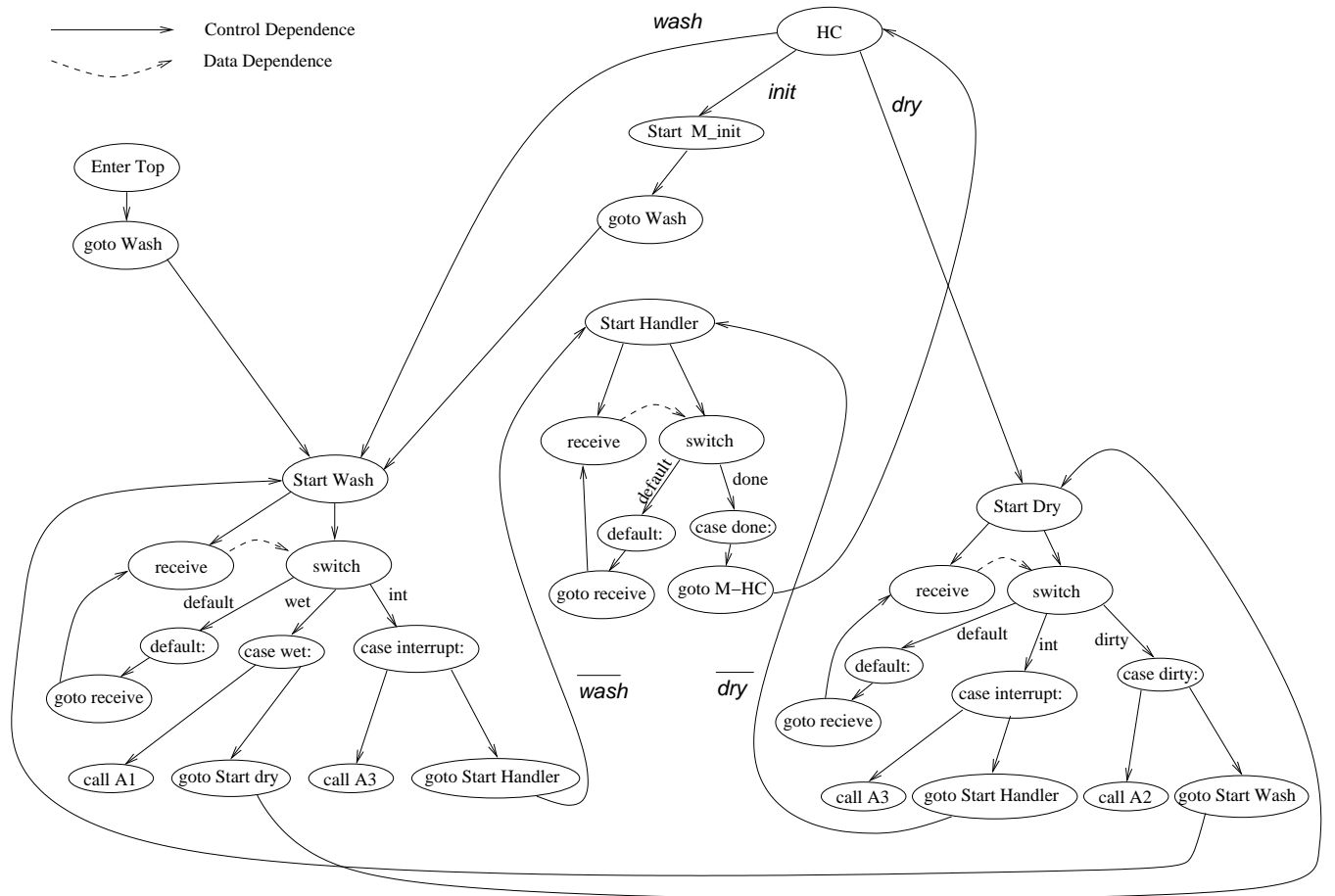


Figure 11: PDG for the History Example

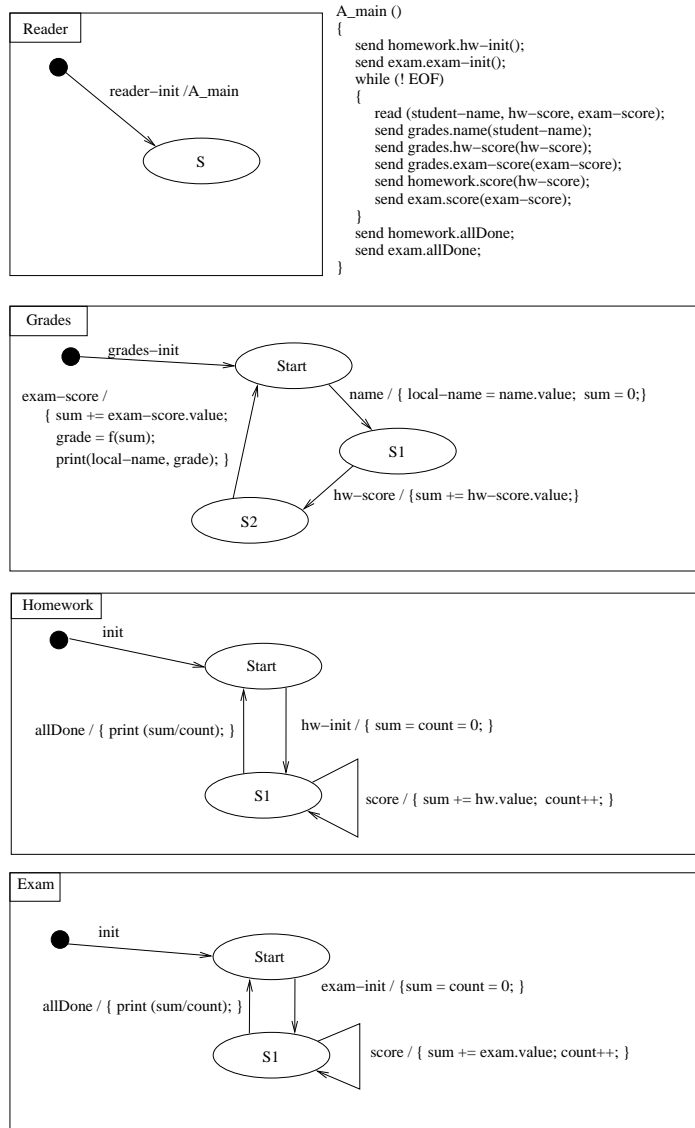


Figure 12: State Diagrams for the Student Grades Example

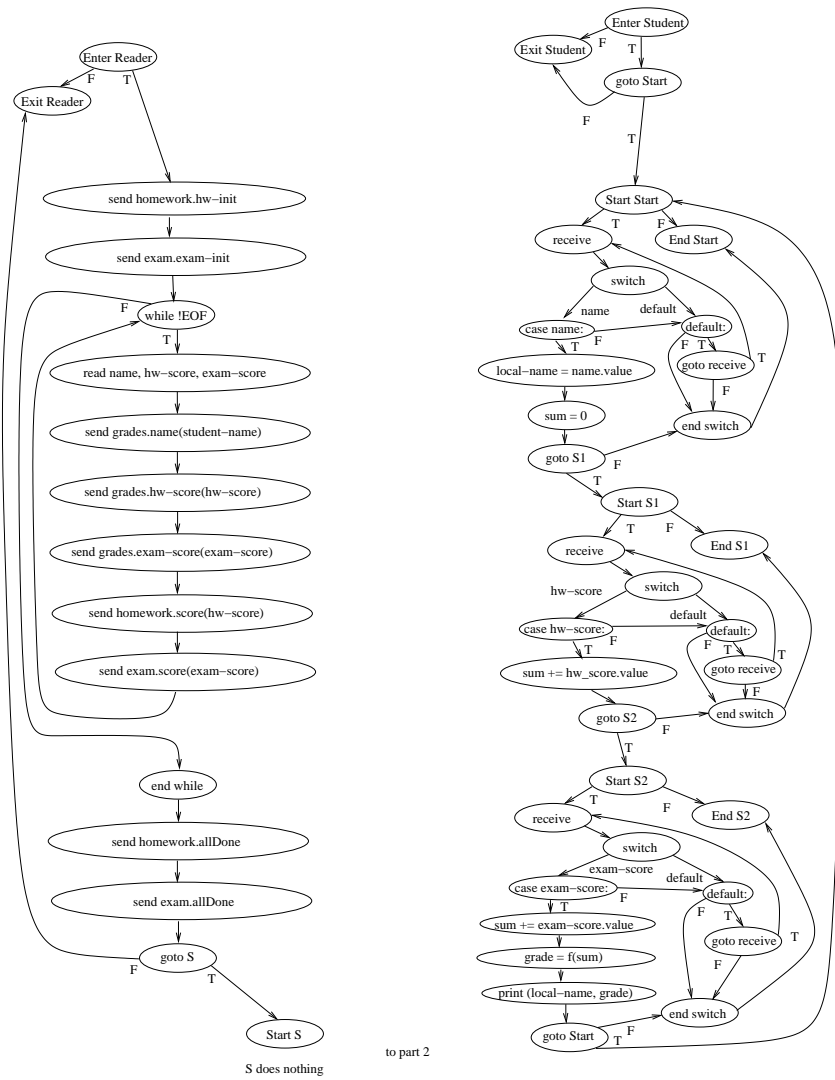


Figure 13: CFG (part 1) for the Student Grades Example

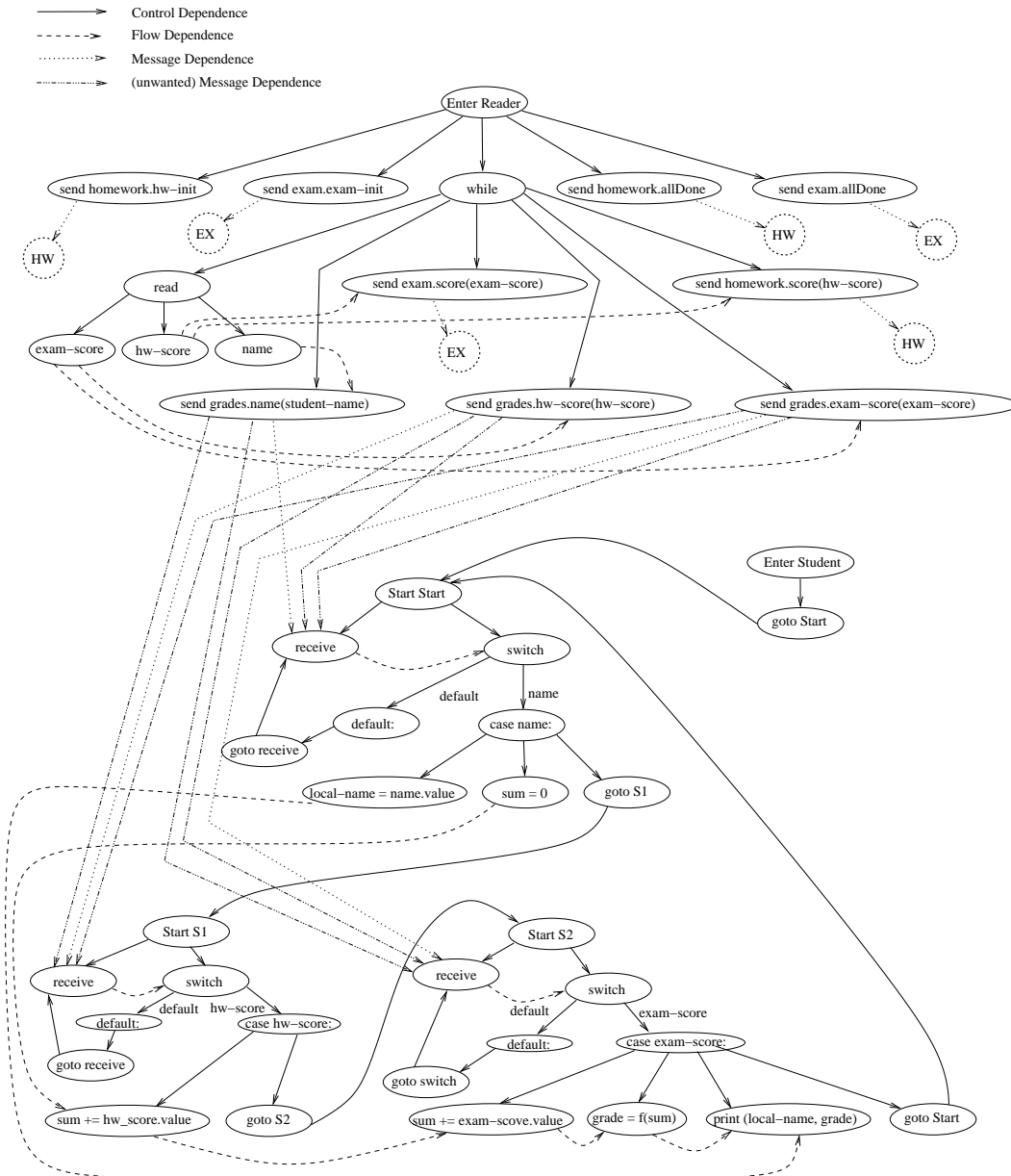


Figure 14: PDG (part 1) for the Student Grades Example

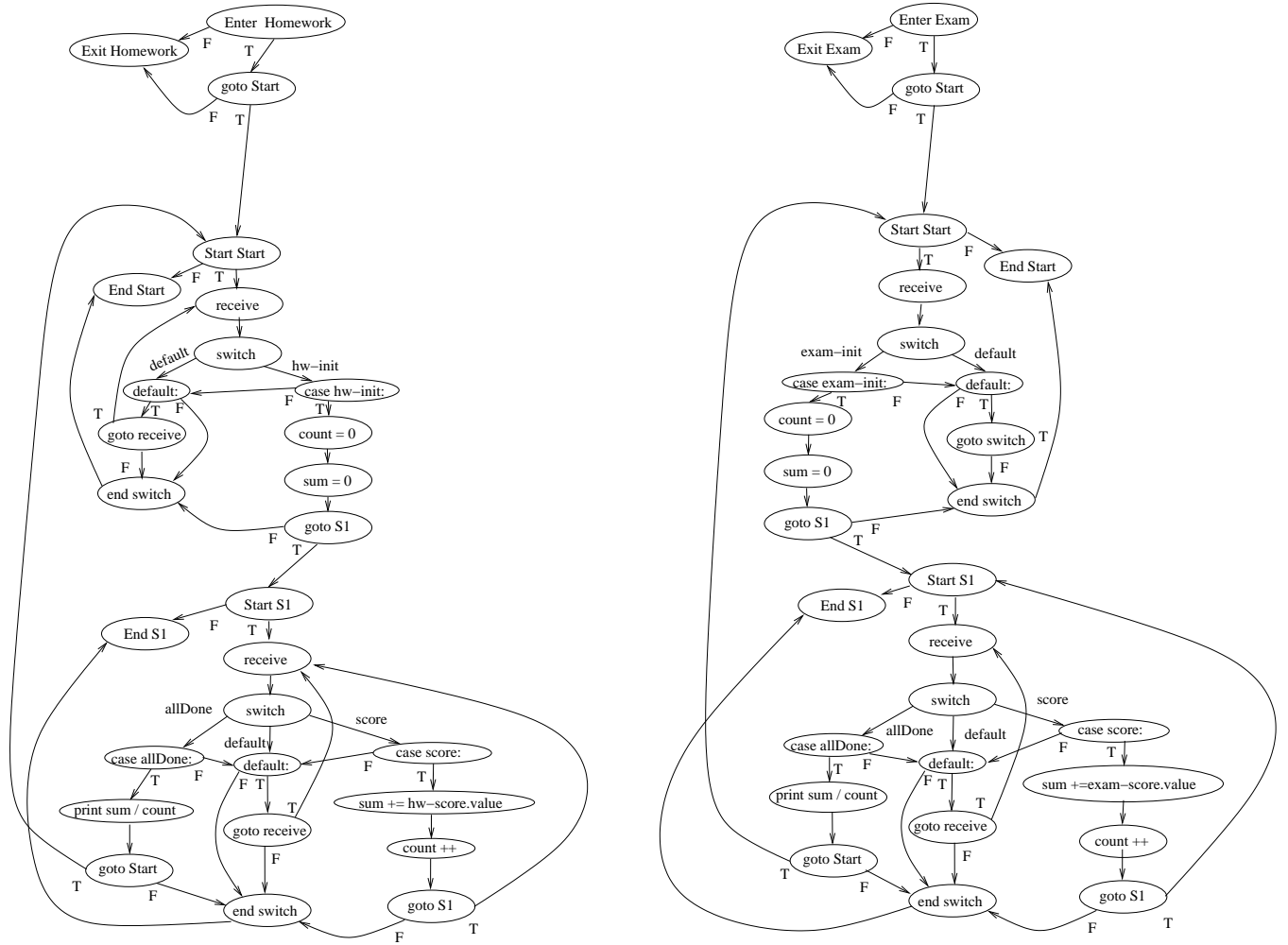


Figure 15: CFG (part 2) for the Student Grades Example

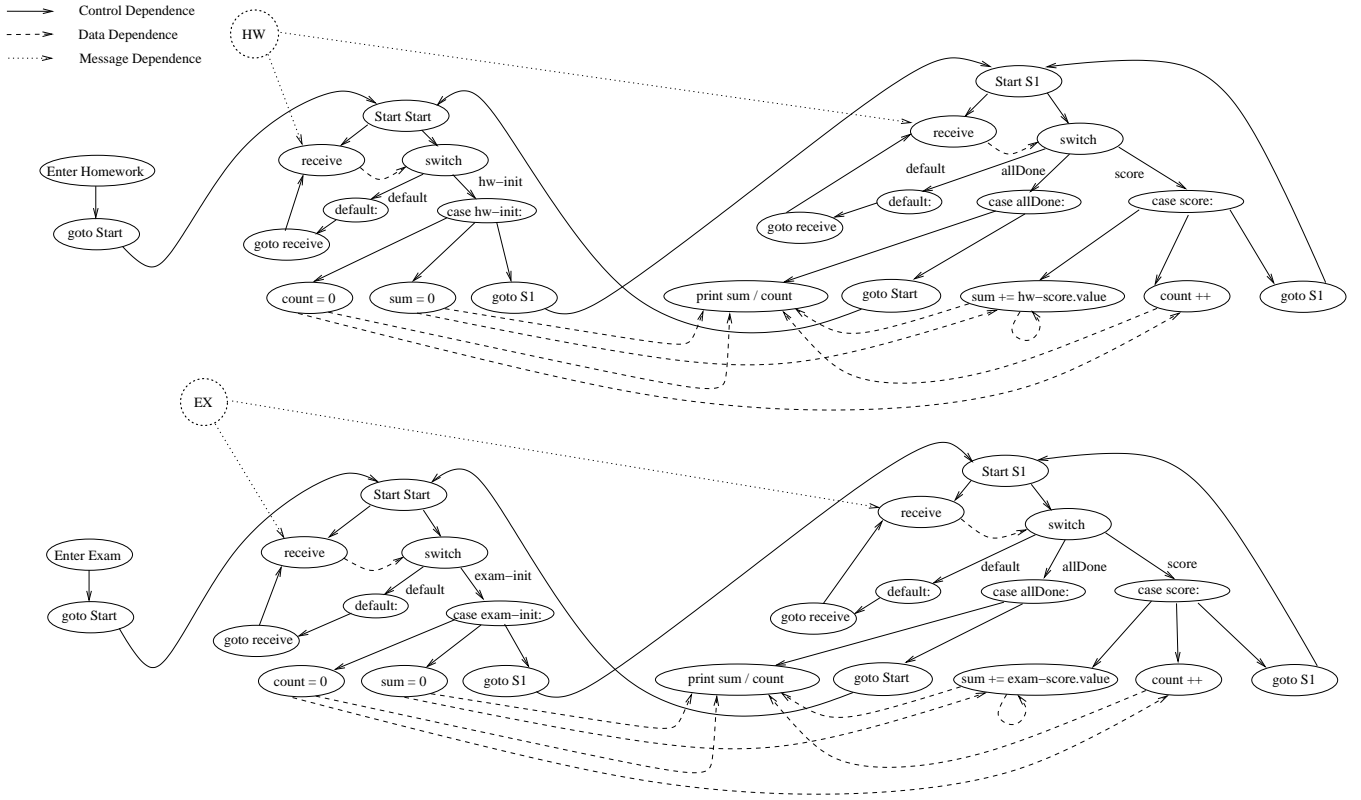


Figure 16: PDG (part 2) for the Student Grades Example

The dotted and dot-dashed edges in the PDGs (Figures 14 and 16) are message-dependence edges, which connect sends and receives. The structure of the *main* and *student grades* machines are such that not all sends can reach all receives (and similarly for some of the other messages). Those that *are* possible are shown using dotted edges, while those that are *not* possible are shown using dot-dashed edges. Assuming that messages do not get lost (and thus the machines do not get out of sync) it should be possible to identify those message dependences that are indicated using dot-dashed edges using static analysis, and thus to omit them from the PDG. (Note that to avoid clutter in Figures 14 and 16 message-dependence edges to the second part of the PDG are shown going to and coming from small dashed circles.)

This examples also illustrates that slicing of Rose specifications is viable. Hand-crafted C code was used to produce a PDG, which was then used to perform two slices: As desired, the slice from “sum” in the *exam* FSM machine does not include the *homework* machine, and the slice from the *count* variable does not include any values, just the calls.

5 Open Questions

5.1 Slicing FSMs

In the case of procedural programs, the goal of a backward slice from a component S has been defined to be:

- To include all components of the program that might affect the execution of S [8].
- To produce a projection of the program so that (barring non-termination) S is executed the same number of times, and the variables used at S have the same sequences of values, in the original program and in the projection [7].
- To identify all components on which S is semantically dependent [6].

Defining a similar goal for a slice of a Rose/RT specification is an open question. There are several aspects to consider, including which FSM paths should be included in a slice, and which transition labels should be included in a slice.

For example, consider computing a slice from some component of a FSM state S in a Rose/RT structure diagram. How much of the rest of the FSM should be included in that slice? In many cases, FSMs are strongly connected (or close to being strongly connected), so a naive approach that follows paths backward in the FSM might always include the entire FSM in every slice. A better possibility might thus be to include only *acyclic* paths from the initial state to state S .

Similarly, when a FSM transition is included in a slice, should the labels on that transition (its trigger, guard, and action) also be included? If the trigger is included, then should the slice also include all related send actions (from other FSMs)? Treating triggers this way is similar to treating them like predicates in a normal program (where all of a node’s control ancestors get included in its slice). We might prefer to think of the path from start to S like the (straight-line) path in a CFG from the CFG’s *enter* node to a node n . In that case, the nodes along that path are included in the slice only if there is data dependence involved.

When slicing with respect to action A of state S , other possibilities are to include only the trigger associated with A , or those triggers that have some parameters used in A .

Note that some of these decisions will determine whether slices of Rose/RT specifications are themselves legal specifications (for example, since transitions must be labeled with triggers, not just with actions, leaving some triggers out of a slice can lead to a slice that is an incomplete specification).

5.2 Slicing Concurrent Programs

Rose/RT specifications involve messages being sent from one FSM to another. Also, the states of a FSM in a structure diagram can include arbitrary local variables, including pointers, and those

pointers can be passed in messages as parameters to other FSMs (other processes). This means that slicing a Rose/RT specification has some issues in common with slicing concurrent programs, which is a problem that has been studied in the past [1, 4, 9, 5, 3, 2].

One approach to slicing concurrent programs is to include an *interference dependence edge* between every write to a variable x that might execute in one thread, and every read of x that might execute in a parallel thread. However, this approach essentially assumes that all paths in the CFG can be taken and that all possible interleavings of parallel threads are possible. The following example illustrates how, in the absence of this assumption, some interference edges do *not* represent possible “flow” of values:

```
y = 0
x = 0
cobegin {
  // thread 1
  print y
  x = 1
  print y
}{
  // thread 2
  if (x == 1)
    y = 5
} coend
```

There are interference edges from “ $y = 5$ ” in Thread 2 to the two “`print y`”s in Thread 1. However, in fact there is no dependence from “ $y = 5$ ” to the first “`print y`” because the assignment to y only happens if x has been set to one, and that only happens *after* the first “`print y`”.

The open question is how to do a better job of identifying interference dependences in the CFG derived from a Rose/RT specification.

References

- [1] J. Cheng. Slicing concurrent programs. *Lecture Notes in Computer Science*, 749:223–240, Nov. 1993.
- [2] B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Lecture Notes in Computer Science*, volume 1694. Springer-Verlag, Sept. 1999.
- [3] M. Dwyer, J. Corbett, J. Hatcliff, S. Sokolowski, and H. Zheng. Slicing multi-threaded java programs: A case study. Technical Report KSU CIS TR 99-7, Kansas State University, 1999.
- [4] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Tools and Software Eng.*, June 1998.
- [5] M. Nanda and S. Ramesh. Slicing concurrent programs. Aug. 2000.
- [6] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. on Software Engineering*, 16(9):965–979, 1990.
- [7] T. Reps and W. Yang. The semantics of program slicing and program integration. In *Lecture Notes in Computer Science*, volume 352, New York, NY, Mar. 1989. Springer-Verlag.
- [8] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.
- [9] J. Zhao. Slicing concurrent java programs. In *Proc. Seventh IEEE Int. Workshop on Program Comprehension*, pages 126–133, May 1999.

A UML-RT Model of the NRL Pump^{*}

GrammaTech

317 N. Aurora Street
Ithaca, NY 14850

1. Introduction

This document presents a brief overview of a executable design model of the NRL Pump[1, 2] using Rational Rose for RealTime (Rose RT). The NRL pump is a device used to control the flow of information between high and low security networks. By controlled we mean that information is allowed to flow from the low security network to the high security network, but not from the high side to the low side. The behavior of the NRL Pump was modeled after the Statemate design presented in [3]. Rose RT is a UML tool with extensions to support the modeling of real-time systems. The extensions to Rose RT are based of Bran Selic's ROOM methodology.

The remainder of this report is divided into the three sections. Section 2 describes the UML RealTime extensions in RoseRT. Section 3 gives a brief overview of the major components of the NRL Pump. Section 4 describes differences between the Statemate model and the Rose RT model. Appendix A contains the class diagram from the Rose RT model of the NRL Pump.

2. UML Extensions in Rose RT

Rose RT presents four new major modeling elements as extensions to UML: Capsules, Ports, Protocols, and Connectors. A thorough description of these elements can be found in [4].

Capsules are the fundamental modeling element in Rose RT. A capsule can be considered as the equivalent of a class that is also a light-weight thread. Message passing is the sole means of communication between capsules. Messages are sent and received through ports. *Ports* are objects whose purpose is to send and receive messages to and from capsule instances. The messages that may be sent or received on a given port are restricted by the port's protocol. A *Protocol* is the set of messages exchanged between two objects. It is basically a contractual agreement defining the valid types of messages that can be exchanged between the participants in the protocol. As an option, a state machine may be used to specify the valid communication sequences for a protocol. Capsules are linked together with connectors. A *connector* is used to connect the ports of two capsule instances. The connector serves to link up partners in communication.

3. NRL Pump Overview

Using [3] as a guide our model of the pump has five key capsules: *NRLPump*, *Mediate_Hi_Unit*, *Mediate_Low_Unit*, *Relay_H_Trans*, and *Relay_L_Trans*. Each of these is briefly described below.

^{*} This research was partially supported by the Defense Advanced Research Projects Agency under Contract F30602-00-C-0080 and by the Office of Naval Research under Contract N00014-99-C-0035.

- *NRLPump*: This capsule models that NRL pump device. There is no behavior modeled for this capsule, its behavior has been instead modeled in sub-capsules. It consists of two key capsules *Mediate_Hi_Unit* and *Mediate_Low_Unit*, and another capsule to model a queue (the reasons for modeling the queue as a capsule are explained in section 4).
- *Mediate_Hi_Unit*: The *Mediate_Hi_Unit*'s primary function is to control the rate of message flow from the queue to the high relay based on feedback from the high relay. After each ACK is received, the mean ACK time is adjusted.
- *Mediate_Low_Unit*: The *Mediate_Low_Unit* enqueues messages from the low relay if there is sufficient space in the buffer. If no space is available, it sleeps until it is informed of a vacancy. It also sends ACKs to the low relay unit exponentially delayed, with parameter equal to the mean ACK time from the *Mediate_Hi_Unit*.
- *Relay_L_Trans*: *Relay_L_Trans* provides the low security interface to the pump. It relays messages to the *Mediate_Low_Unit* and returns ACKs to the low security network when directed by the *Mediate_Low_Unit*.
- *Relay_H_Trans*: *Relay_H_Trans* provides the high security interface to the pump. It accepts messages from the *Mediate_Hi_Unit* and relays them to the high security network.

4. Key Differences Between Model Versions

Using different modeling tools and methodologies results in minor differences in behaviors of the models. The two key sources of differences in the model are the differences in the definition of what can trigger a transition in a state machine and the fact that capsules can only communicate via message passing.

In the Statemate version of the model transitions in the state machine are often triggered by the reading and writing of variables. As all communication must be done by message passing, events based on another capsule reading a variable are obviously not allowed. In fact all transition triggers are based on the reception of a message. This forced us to build in explicit messages when reading and writing to variables was to have triggered a transition. Most notable example of this is the pump's buffer. We modeled this as a separate capsule called Queue. The queue used in this pump model has three states: empty, full, and not empty or full. The primary operation of the queue is to take messages from the low mediator and make them available to the high mediator. Items were added and removed from the queue by sending messages to the queue.

5. References

1. Kang, M.H. and I.S. Maskowitz, *A Pump for rapid reliable, secure communication*. in *ACM Conference on Computer and Communications Security*. 1993.
2. Kang, M.H., I.S. Moskowitz, and D.C. Lee, *A Network Pump*. 1997, Naval Research Laboratory.
3. Moore, A.P., *A Behavioral requirements model for the NRL Pump*. in *NRL Technical Memorandum 5540-021a:apm*. 1995. Washington, D.C.
4. Rational, *Rose For RealTime Help Pages*. .

Appendix A

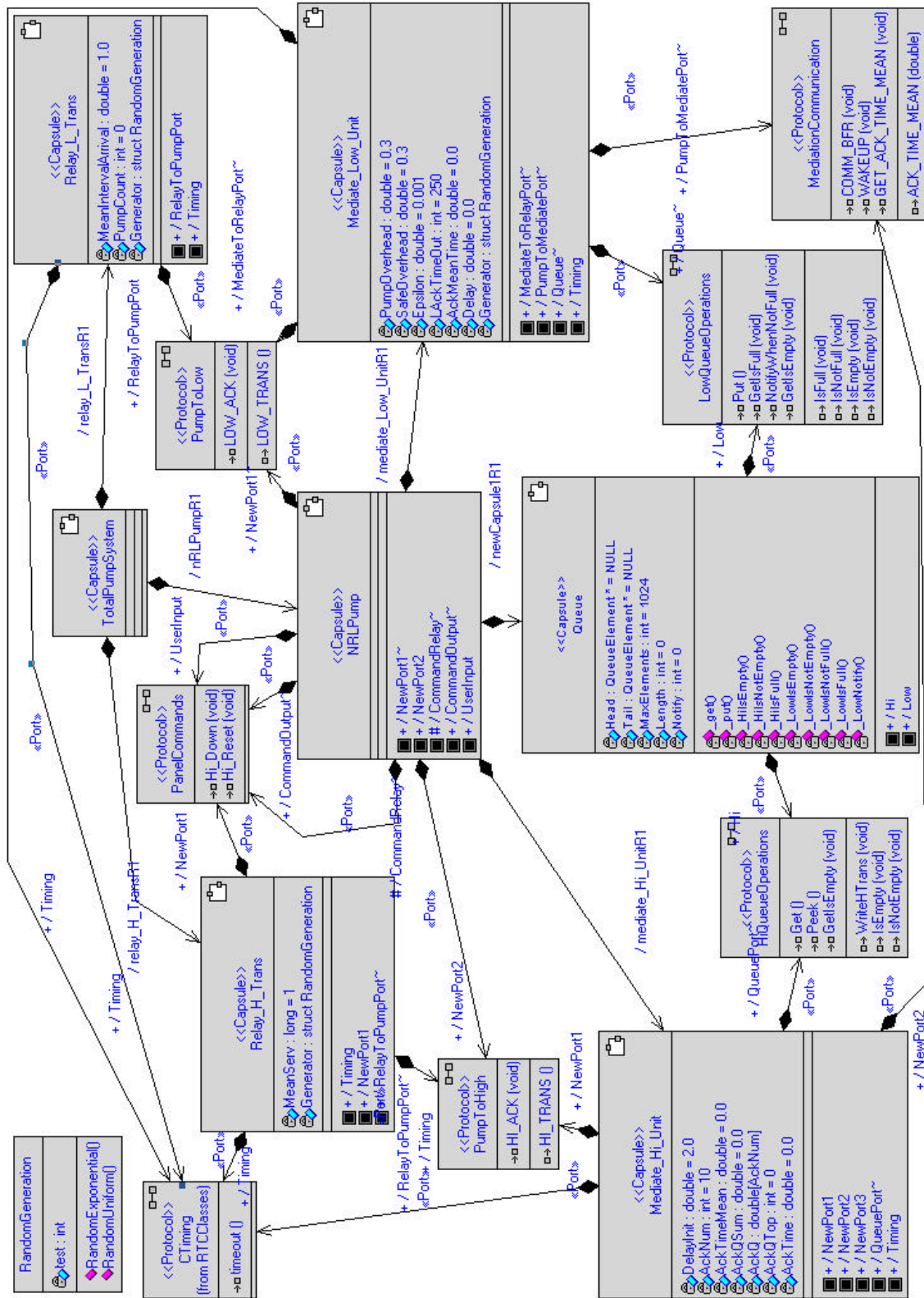


Figure 1 NRL Pump Class Diagram

Better Slicing of Programs with Jumps and Switches

Sumit Kumar¹ and Susan Horwitz^{1,2}

¹University of Wisconsin

²GrammaTech, Inc.

Abstract

Program slicing is an important operation that can be used as the basis for programming tools that help programmers understand, debug, maintain, and test their code. This paper extends previous work on program slicing by providing a new definition of “correct” slices, by introducing a representation for C-style switch statements, and by defining a new way to compute control dependences and to slice a program-dependence graph so as to compute more precise slices of programs that include jumps and switches. Experimental results show that the new approach to slicing can sometimes lead to a significant improvement in slice precision.

1 Introduction

Program slicing, first introduced by Mark Weiser in [11], is an important operation that can be used as the basis for programming tools that help programmers understand, debug, maintain, and test their code. Slicing was defined by Weiser as the solution to a dataflow problem specified using the program’s control-flow graph (CFG). Ottenstein and Ottenstein [8] provided a more efficient algorithm (reviewed below in Section 2.1) that uses the program-dependence graph (PDG) [4].

This paper makes the following four contributions in the area of program slicing:

Defining Correct Slices: Weiser defined a *correct* slice of a program P to be a projection of P with certain properties (see Section 3). Podgurski and Clarke [9] defined a notion of *semantic dependence* that can also be used as the basis for a definition of a correct slice; however, their definition did not take jump statements (`goto`, `break`, etc.) into account.

We give an example to illustrate a shortcoming of Weiser’s definition, and offer a new definition, similar to the one for semantic dependence, that overcomes the problem with Weiser’s definition, and also makes sense for programs with jump statements.

Language Extension: We discuss how to represent C-style switch statements in the CFG and the PDG. To our knowledge, this is the first time switch statements have been discussed as such, rather than assuming that they have been implemented at a low level using `gotos`. Handling switch statements is important because many slicing applications involve displaying the result of a slice to the programmer, or using the results to create new source code. Thus, for those applications, if a slice includes code from a switch, it needs to be displayed / represented in the new code as a switch rather than in some low-level form. Representing and slicing a switch in a low-level form and then mapping the results back to the source level may lead to a final result that is less precise than the one produced by working on the switch directly.

Improved Precision: Finding correct, minimal slices is an undecidable problem, whether correctness is defined according to Weiser, Podgurski/Clarke, or using the new definition proposed here. However, it is still a reasonable goal to design a slicing algorithm that is more precise than previous ones; i.e., to define a new algorithm that is correct, and also produces smaller slices than previous algorithms.

In this spirit, we introduce some example programs with jumps and switches for which previous slicing algorithms produce slices that include too many components. While the examples with jumps are somewhat artificial, the examples with switches are motivated by code from real programs. We show that the reason “extra” components are included in the slices has to do both with how control dependences are defined, and how slices are computed. We then give a new definition of control dependence and a new slicing algorithm that is more precise than previous algorithms in the presence of jumps and/or switches.

Experimental Results: While it is possible to produce artificial examples in which our new approach to slicing provides arbitrarily smaller slices than previous approaches, it is important to know how well it will work in practice. We provide some experimental results that show that while in most cases slice sizes are reduced by no more than 5%, there are examples of reductions of up to 35%.

2 Background

2.1 Slicing using the PDG

Informally, the slice of a program from statement S is the set of program components that might affect S , either by affecting the value of some variable used at S , or by affecting whether and how often S executes. More precise definitions have been proposed, and are discussed below in Section 3.

Slicing was originally defined by Weiser [11] as the solution to a dataflow problem specified using the program’s control-flow graph (CFG). Ottenstein and Ottenstein [8] provided a more efficient algorithm that uses the

program-dependence graph (PDG) [4]:

Algorithm 1 (*Ottensteins’ algorithm for building and slicing the PDG*)

Step 1: *Build the program’s CFG, and use it to compute data and control dependences: Node N is **data dependent** on node M iff M defines a variable x , N uses x , and there is an x -definition-free path in the CFG from M to N . Node N is **control dependent** on node M iff N postdominates one but not all of M ’s CFG successors.¹*

Step 2: *Build the PDG. The nodes of the PDG are almost the same as the nodes of the CFG: a special enter node, and a node for each predicate and each statement in the program; however, the PDG does not include the CFG’s exit node. The edges of the PDG represent the data and control dependences computed using the CFG.*

Step 3: *To compute the slice from statement (or predicate) S , start from the PDG node that represents S and follow the data- and control-dependence edges backwards in the PDG. The components of the slice are all of the nodes reached in this manner.*

Example: Figure 1 shows a program that computes the product of the numbers from 1 to 10, its CFG, and its PDG. The nodes in the slice of the PDG from “**print(k)**” are shown using bold font. (For the purposes of control-dependence computation, an edge is added to the CFG from the *enter* node to the *exit* node; to avoid clutter, those edges are not shown in the CFGs given in this paper).

2.2 Handling Jumps

Early slicing algorithms (including Weiser’s and the Ottensteins’) assumed a structured language with conditional statements and loops, but no jump statements (such as **goto**, **break**, **continue**, and **return**). Both [2] and [3] pointed out that if a CFG is used in

¹By definition, every CFG node postdominates itself. Thus, if a node M has CFG successors m_1 and m_2 , then unless m_1 postdominates m_2 , m_1 is control-dependent on M (and similarly for m_2).

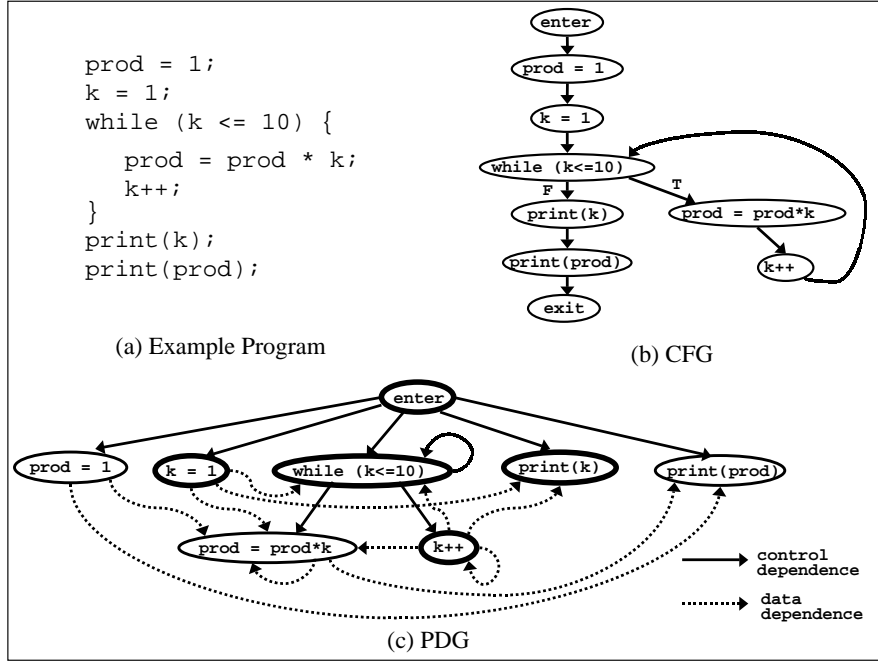


Figure 1: Example program, its CFG, and its PDG. The PDG nodes in the slice from “**print(k)**” are shown in bold.

which a jump statement is represented as a node with just a single outgoing edge (to the target of the jump), then no other node will be control dependent on the jump, and thus it will not be in the slice from any other node. For example, Figure 2(a) shows a modified version of the program from Figure 1, now including a **break** statement. Figures 2(b) and 2(c) show the program’s CFG and the corresponding PDG. Note that in this PDG, there is no path from the **break** to “**print(k)**” or to “**print(prod)**”, and therefore the **break** is (erroneously) not included in the slices from those two print statements even though the presence of the **break** can affect the values that are printed.

The solution proposed by [2] and [3] involves using an augmented CFG, called the ACFG, to build a dependence graph whose control-dependence edges are different from those in the PDG used by Algorithm 1. We will refer to the new dependence graph as the APDG, to distinguish it from the PDG.

Algorithm 2 (*Building and Slicing the*

APDG)

Step 1: *Build the program’s ACFG. In the ACFG, jump statements are treated as pseudo-predicates. Each jump statement is represented by a node with two outgoing edges: the edge labeled true goes to the target of the jump, and the (non-executable) edge labeled false goes to the node that would follow the jump if it were replaced by a no-op. Labels are treated as separate statements; i.e., each label is represented in the ACFG by a node with one outgoing edge to the statement that it labels.*

Step 2: *Build the program’s APDG. Ignore the non-executable ACFG edges when computing data-dependence edges; do not ignore them when computing control-dependence edges. (This way, the nodes that are executed only because a jump is present, as well as those that are not executed but would be if the jump were removed, are control dependent on the jump node, and therefore the jump will be in-*

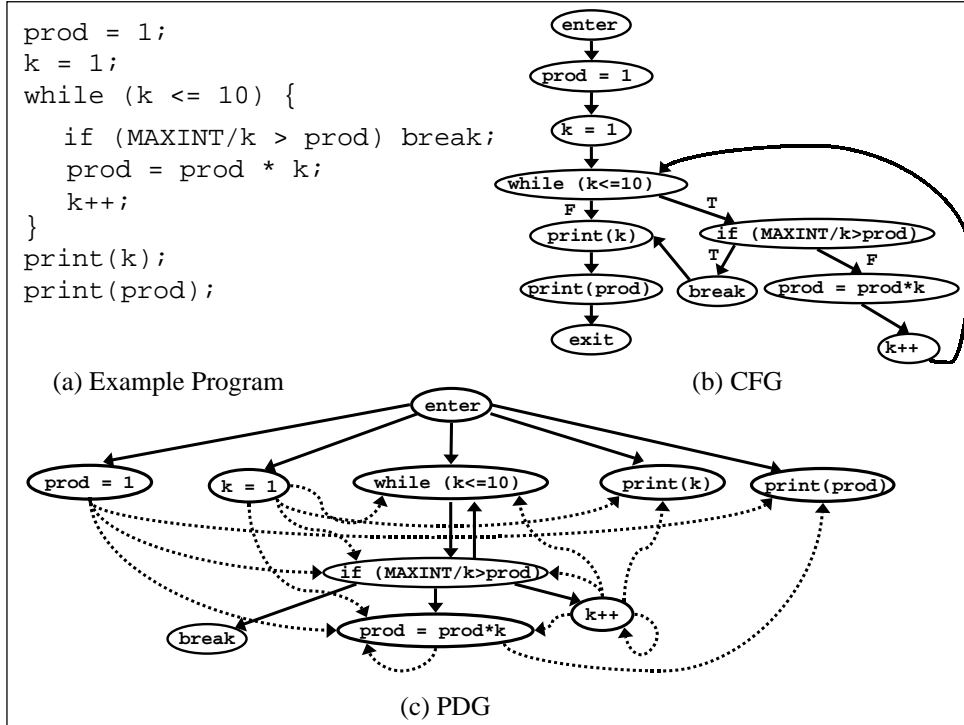


Figure 2: Example program with a `break` statement, its CFG, and its PDG.

cluded in their slices.)

Step 3: To compute the slice from node S , follow data- and control-dependence edges backwards from S as in Algorithm 1. A label L is included in a slice iff a statement “goto L ” is in the slice.

Example: Figure 3 shows the ACFG for the program in Figure 2(a), and the corresponding APDG. (The non-executable *false* edge out of the `break` in Figure 3(a) is shown using a dotted arrow.) Note that in Figure 3(b), there are control-dependence edges from the `break` to “`prod = prod * k`” and to “`k++`”; therefore, the `break` is (correctly) included in every slice that includes one of those two nodes.

3 Semantic Foundations for Slicing

In his seminal paper on program slicing [11], Weiser defined a slice of a program P from point S with respect to a set of variables V to be any program P' such that:

- P' can be obtained from P by deleting zero or more statements.
- Whenever P halts on input I , P' also halts on input I , and the two programs produce the same sequences of values for all variables in set V at point S if it is in the slice, and otherwise at the nearest successor to S that is in the slice.

One problem with this definition is that it can be inconsistent with the intuitive idea that the slice of a program from point S is the set of program components that might *affect* S . For example, Figure 4 shows a program, the slice that a programmer would probably produce if asked to slice the program from statement [6] with respect to variable z , and another slice that is correct according to Weiser’s definition, but that does not match our intuition about slicing. Furthermore, the requirement that a slice be an executable program may be too restrictive in some contexts (e.g., when using slicing to understand how a program works, or to understand the effects of a proposed change). In those cases, it might be more appropriate

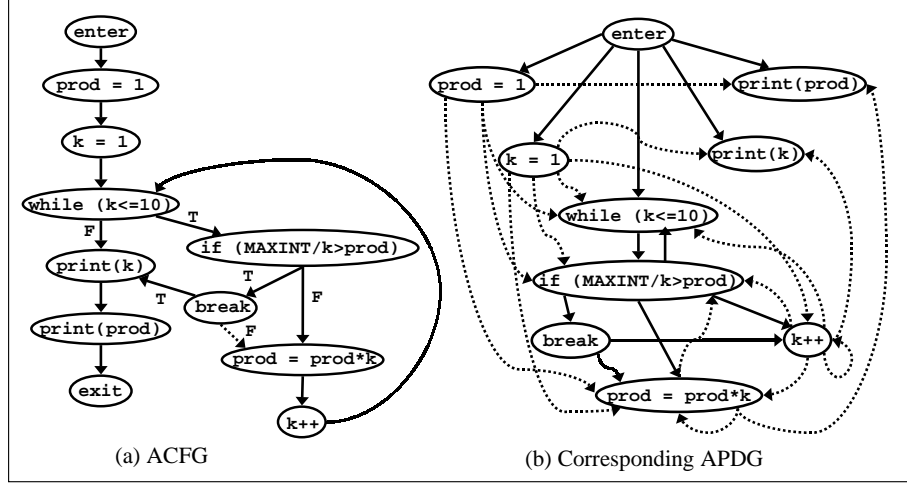


Figure 3: ACFG and the corresponding APDG for the example with a break.

to define the slice of a program simply to be a subset of the program’s components, rather than an executable projection of the program.

Given these observations, we propose to define the slice of program P from component S to be the components of P that might have a semantic effect on S . But what does it mean for a statement or predicate X to have a semantic effect on another statement/predicate S ? To make that notion more precise, we consider what happens when a new program P' is created by modifying X or removing it from program P as follows:

X is a normal predicate: P' is created by replacing X with a different predicate that uses the same set of variables as X . (For example, in the program whose ACFG is shown in Figure 3, the predicate “ $\text{MAXINT}/k > \text{prod}$ ” could be replaced by any other predicate that uses only variables k and prod , such as: “ $k < \text{prod}$ ”, or “ $k \neq 0 \ \&\& \ \text{prod} > 22$ ”.)

X is a pseudo-predicate (a jump statement): P' is created by removing statement X from P .

X is a non-jump statement: P' is created by replacing X with a different statement that uses and defines the same sets of variables as X . (For example, in the program whose ACFG is shown in Figure 3, the statement “ $\text{prod} = \text{prod} * k$ ” could be replaced by any other statement that uses only variables prod and k , and that defines variable prod , such as: “ $\text{prod} = k + \text{prod}$ ”, or “ $\text{prod} = \text{prod} - k - 4$ ”.)

Definition 0 (Semantic effect): X has a *semantic effect* on S iff there is some program P' created by modifying or removing X from P as defined above, and some input I such that:

- Both P and P' halt on I .
- The two programs produce a different sequence of values for some variable used at S .

Note that the sequence of values produced for a variable used at S can differ either because the two sequences are of different lengths, or because their k^{th} values differ (for some k).

Program	Intuitive Slice	Also Correct by Weiser's Definition
<pre> [1] x = 2; [2] y = 2; [3] w = x * y; [4] x = 1; [5] y = 3; [6] z = x + y; </pre>	<pre> [4] x = 1; [5] y = 3; [6] z = x + y; </pre>	<pre> [1] x = 2; [2] y = 2; [6] z = x + y; </pre>

Figure 4: Example illustrating a shortcoming of Weiser's definitions of a correct slice.

Definition 0 is similar to the definition of *finitely demonstrable semantic dependence* given by Podgurski and Clarke in [9]. However, that definition did not take jump statements into account: according to their definition, no program component is ever semantically dependent on a jump; therefore, if a correct slice from S is defined to include all components on which S is semantically dependent, jump statements will never be included in a slice. This is clearly contrary to one's intuition, and therefore is a shortcoming of the Podgurski/Clarke definition.

As usual with any interesting property of a program, determining which components have a semantic effect on a given component S , according to Definition 0, is an undecidable problem. Therefore, we must say that a (correct) slice of program P from component S is any superset of the components of P that have a semantic effect on S .

Note that using Definition 0, statements [4] and [5] in the example program in Figure 4 (but not statements [1] and [2]) have a semantic effect on statement [6]. Therefore, a correct slice from statement [6] must include statements [4] and [5] (but not statements [1] and [2]), which is consistent with our intuition about that slice.

4 Representing Switch Statements

Consider a C switch statement of the form:

```
switch (E) {
```

```

    case e1: S1; break;
    case e2: S2; break;
    ...
    case en: Sn; break;
    default: S;
}
```

Clearly, “**switch**(E)” should be represented in the CFG (and the ACFG) using a (normal) predicate node with $n + 1$ outgoing edges: one to each **case** including the **default**. If there were no **default**, the $n + 1^{st}$ edge should go to the first statement following the switch (because in C, if the value of the switch expression does not match any case label, and there is no **default** then execution continues immediately after the switch).

Now consider how to represent the case labels. One's initial intuition might be that they are similar to other labels in a program (the targets of **goto** statements). However, there is an important difference: if a program includes “**goto** L1”, then label L1 must be in the program, or it is not syntactically correct. If there is *no* “**goto** L1”, then it doesn't matter whether label L1 is in the program: its presence or absence has no semantic effect. However, these observations are not true of a case label. Removing a case label from a program never causes a syntax error, but *can* have a semantic effect. For example, if expression E in the code given above evaluates to **e2**, then statement S2 will execute. However, if “**case e2**” is removed, then statement S2 will not execute; instead, statement S will execute. Therefore, it makes sense for “**case e2**” to be in the slice from S2 as well as in the slice from S.

This suggests that, like jumps, case labels should be represented using pseudo-predicates in the ACFG. The target of the outgoing *true* edge from a case-label node should be the first statement inside the **case**, and the target of the outgoing *false* edge should be the node that represents the default label if there is one, and otherwise the first statement that follows the switch (because if the case label is removed, and the switch expression matches that value, then execution proceeds with the first statement after the switch). The target of the outgoing *false* edge from the default case should always be the first statement that follows the switch.

Example: Figure 5 shows the ACFG for the switch statement given above (for $n = 3$).

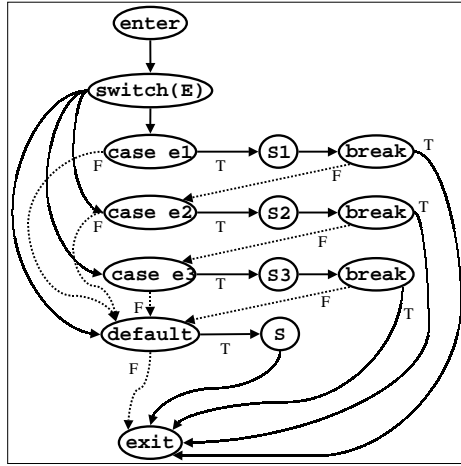


Figure 5: ACFG for a switch statement.

5 Motivation for a New Slicing Algorithm

Figure 6 gives three examples where Algorithm 2 (see Section 2.2) produces slices that include unwanted components. (In these examples, we assume that switch statements are represented in the ACFG as discussed above in Section 4.) The first column in Figure 6 gives a code fragment, with one statement enclosed in a box. The second column shows the ideal slice from the boxed statement (according to Definition 0 given above in Section 3). The third column shows the slice computed using

Algorithm 2. The first two examples involve switches, while the third example involves only gotos.

Note that in the first example the slice from **S3** should include the **break** from the previous case, because the presence/absence of that **break** affects whether or not **S3** executes. In particular, consider what happens when expression **E** evaluates to **e2**. If the **break** is *not* in the program, **S3** executes, while if the **break** *is* in the program, **S3** does not execute.

In the second example, the slice from **S** should include neither “**if** (**P**)” nor “**return**”. Whatever the value of predicate **P**, statement **S** will not execute (because either the **return** or the **break** prevents execution from “falling through” from “**case e1**” to “**case e2**”). Similarly, whether or not the **return** is in the program makes no difference since it is followed by the **break** (and thus **S** is always prevented from executing).

In all three examples, extra components are included in the slices computed using Algorithm 2 because of a chain of control-dependence edges. For instance, the APDG for the second example includes the following chain: **case e1** \rightarrow **if** (**P**) \rightarrow **return** \rightarrow **break** \rightarrow **case e2** \rightarrow **S**. Thus, since Algorithm 2 follows all control-dependence edges backwards, all of those components are included in the slice from **S**².

In this example, each individual control-dependence edge represents a possible semantic effect: “**case e1**” has a semantic effect on “**if** (**P**)”, which has a semantic effect on “**return**”, which has a semantic effect on “**break**”, which has a semantic effect on “**case e2**”, which has a semantic effect on **S**. However, the backwards closure of the control-dependence relation starting from **S** yields a superset of the components that have a semantic effect on **S**; i.e., the “semantic-effect” relation is not transitive.

It is also possible to have an example in which even an individual control dependence (computed using the ACFG) does not reflect a semantic effect, as illustrated in Figure 7. In

²Furthermore, the entire backward closure from predicate **P** of the control- and data-dependence relations will be included in the slice computed by Algorithm 2, making it arbitrarily larger than the ideal slice.

Code Fragment	Ideal Slice	Slice computed using Algorithm 2
<pre>switch (E) { case e1: S1; break; case e2: S2; break; case e3: S3; break; }</pre>	<pre>switch (E) { case e1: S1; break; case e2: S2; break; case e3: S3; break; }</pre>	<pre>switch (E) { case e1: S1; break; case e2: S2; break; case e3: S3; break; }</pre>
<pre>switch (E) { case e1: if (P) return; break; case e2: S; }</pre>	<pre>switch (E) { case e1: if (P) return; break; case e2: S; }</pre>	<pre>switch (E) { case e1: if (P) return; break; case e2: S; }</pre>
<pre>if (P1) goto L1; if (P2) goto L3; goto L2; L1: S; L2: ... L3: ...</pre>	<pre>if (P1) goto L1; if (P2) goto L3; goto L2; L1: S; L2: ... L3: ...</pre>	<pre>if (P1) goto L1; if (P2) goto L3; goto L2; L1: S; L2: ... L3: ...</pre>

Figure 6: Examples for which Algorithm 2 produces slices with extra components.

this example, the APDG includes a control-dependence edge from “if (P)” to S1 because S1 postdominates the *true* successor of the if in the ACFG, but does not postdominate its *false* successor (because the *goto*’s non-executable *false* edge bypasses S1). However, “if (P)” cannot in fact affect the execution of S1; it always executes, regardless of whether P evaluates to *true* or to *false*.

These examples motivate the need for a new definition of control dependence to avoid control-dependence edges like the one in Figure 7 that do not reflect a semantic effect. They also motivate the need for a new way to compute slices that does not involve taking the transitive closure of the control-dependence edges, since, as discussed above, the semantic-effect relation is not transitive.

6 New Definition of Control Dependence and New Slicing Algorithm

Recall that the definition of control dependence used in Algorithm 1 is as follows:

Definition 1 (Original control dependence): Node N is **control dependent** on node M iff N postdominates, in the CFG, one but not all of M ’s CFG successors.

To permit control dependence on jumps, Algorithm 2 replaces “CFG” with “ACFG” in the definition of control dependence:

Definition 2 (Augmented control dependence): Node N is **control dependent** on node M iff N postdominates, in the ACFG, one but not all of M ’s ACFG successors.

Unfortunately, as illustrated in Figure 7, Definition 2 is too liberal; it can cause a spurious control dependence of N on M due to the presence of an intervening pseudo-predicate. For example, in the ACFG in Figure 7, node S1 fails to postdominate the *false* successor of the if only because of the non-executable edge from “goto L1” to S2. Since the execution of S1 is affected by the presence/absence of the *goto* it *should* be considered to be control dependent on the *goto*; however, (as noted previously), S1 will execute regardless of the value of predicate P, and therefore it should *not* be considered to be control dependent on the if. So in this case,

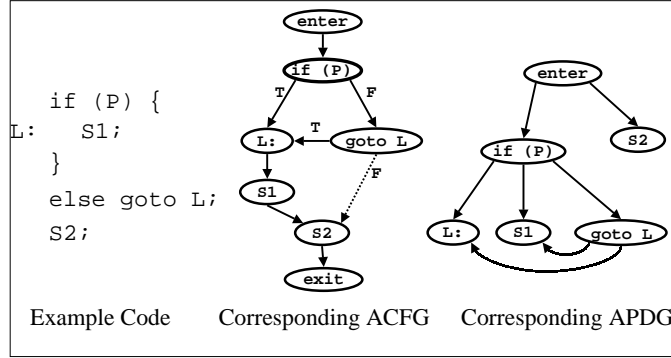


Figure 7: Example in which a control dependence does not reflect a semantic effect.

the actual influence of “goto L1” on statement S1 causes an apparent (but spurious) influence of “if (P)” on S1.

The solution to this dilemma is to replace only the *second* instance of “CFG” with “ACFG” in Definition 1:

Definition 3 (Control dependence in the presence of pseudo-predicates): Node N is **control-dependent** on node M iff N post-dominates, in the CFG, one but not all of M ’s ACFG successors.

We will refer to a dependence graph that includes control-dependence edges computed using Definition 3 as a PPDG (pseudo-predicate PDG) to distinguish them from the PDGs whose control-dependence edges are computed using Definition 1, and the APDGs whose control-dependence edges are computed using Definition 2.

Example: The program and ACFG from Figure 7 are given again in Figure 8, with the corresponding PPDG. Note that neither label L nor statement S1 is control dependent on “if (P)”.

Definition 3 addresses the problem of control-dependence edges that do not reflect semantic effects. The next problem that needs to be addressed is the fact that even when every control-dependence edge does represent a semantic effect, the backward closure of control-

dependence edges from a node S may include nodes that have no semantic effect on S . For example, consider again the PPDG in Figure 8. If the slice from node S1 includes all nodes reached by following control-dependence edges backwards, then “if (P)” will (erroneously) be in the slice because of the chain of control-dependence edges: if (P) \rightarrow goto L \rightarrow S1.

To address this problem, we need the following definition:

Definition 4 (IPD): The **immediate post dominator (IPD)** of a set of ACFG nodes is the node that is the least-common ancestor of that set of nodes in the CFG’s postdominator tree.

Consider a (normal or pseudo) predicate P , with ACFG successors $n_1 \dots n_k$, and let $D = \text{IPD}(n_1 \dots n_k)$. Intuitively, P may affect the execution of a program component S only if there is a path in the CFG from one of P ’s ACFG successors to S that does not include node D . (If there is such a path, we say that S is **controlled by** P .) The value of P (for a normal predicate), or its presence/absence (for a pseudo-predicate) determines which of its ACFG successors is executed. The execution of the nodes along the paths from those ACFG successors to D are also affected by the value (or presence/absence) of P . However, since whenever P is executed, execution will al-

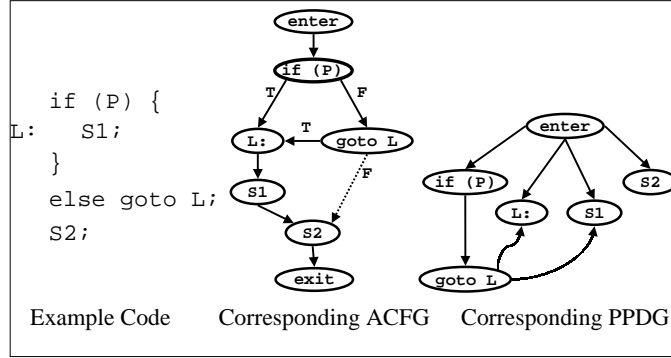


Figure 8: Example from Figure 7 with the corresponding PPDG.

ways reach D (barring an infinite loop or other abnormal termination), the execution of nodes “beyond” D are not affected by P .

As discussed above, following control-dependence edges backwards from S in the PPDG can cause “extra” nodes to be included in the slice from S . In terms of the “is controlled by” relation, this is because there may be a chain of control-dependence edges in the PPDG from a predicate P to S , yet S is not controlled by P . However, we have proved the following Theorem:

Theorem: Node S is controlled by (normal or pseudo) predicate P iff there is a chain of control-dependence edges in the PPDG:

$$P \rightarrow M_1 \rightarrow M_2 \rightarrow \dots \rightarrow M_k \rightarrow S$$

such that every M_i in the chain is a normal predicate node. (Note that there may also be no M_i ’s at all; i.e., there may be a single control-dependence edge $P \rightarrow S$.)

The Theorem tells us that it is not necessary to follow control-dependence edges back from a pseudo-predicate; for any predicate P such that there is a node S in the slice that is controlled by P , P will be picked up by following chains backwards only from normal predicates.

The new algorithm for building and slicing the PPDG is given below.

Algorithm 3 (*Building and slicing the PPDG*)

Step 1: Build the ACFG as described above for Algorithm 2.

Step 2: Build the PPDG: Ignore the non-executable ACFG edges when computing data-dependence edges; compute control-dependence edges according to Definition 3.

Step 3: To compute the slice from node S , include S itself and all of its data- and control-dependence predecessors in the slice. Then follow backwards all data-dependence edges, and all control-dependence edges whose targets are not pseudo-predicates; add each node reached during this traversal to the slice. Include label L in the slice iff a statement “goto L ” is in the slice.

Examples:

1. The slice of the program in Figure 8 computed using Algorithm 3 would include the nodes for $S1$, “goto L ”, L , and the *enter* node. It would not include the node for “if (P)” because, since “goto L ” is a pseudo-predicate, its incoming control-dependence edge would not be followed back to the if node.

- Figure 9 shows the code, ACFG, and PPDG for the second example in Figure 6. Bold font is used to indicate the nodes that would be in the slice from statement **S** computed using Algorithm 3. Note that “**case e1**”, “**if (P)**”, and “**return**” are correctly omitted from the slice.

6.1 Complexity

The time required for Algorithm 3 includes the time to build the PPDG and the time to compute a slice. Like previous slicing algorithms that use a dependence graph, the time for slicing itself is extremely efficient, requiring only time proportional to the size of the slice (the number of nodes and edges in the sub-PPDG that represents the slice).

The only difference in the time required to build the PPDG as compared to the time required to build the APDG is for the computation of control dependences. Computing control dependences can be done for both the APDG and the PPDG in time $O(E+C)$, where E is the number of edges in the ACFG and C is the number of control-dependence edges. However, C may be different for the APDG and PPDG. For example, in Figure 9, the PPDG includes edges from “**switch (E)**” to “**if (P)**” and to **S** that would not be in the corresponding APDG. Figures 7 and 8 illustrate control-dependence edges that are in the APDG but not in the PPDG.

7 Interprocedural Slicing

The Ottensteins’ algorithm for intraprocedural slicing using the PDG was extended to interprocedural slicing using the *System Dependence Graph* (SDG) in [6]. The approach is as follows:

- Use interprocedural dataflow analysis to determine what non-local variables might be used or modified by each procedure; for each procedure and procedure call, add an explicit input parameter for each variable that might be used or modified, and an explicit output parameter for each variable that might be modified.
- Build the PDG for each procedure (including nodes to represent calls, input parameters, and output parameters), and connect the PDGs with edges that represent procedure calls and parameter passing.
- Compute and add *summary edges* to represent the (transitive) effects that each procedure’s input parameters might have on its output parameters. This is done essentially by performing intraprocedural slices from the nodes that represent the final values of the output parameters (an iterative process is used to account for recursion).
- To compute the slice from node S , use a two-phase approach: Phase 1 follows edges backwards from S , including the interprocedural edges that represent calls made *to* S , but not those that represent calls made *from* S . Phase 2 starts from all nodes reached during Phase 1, and follows edges backwards again; this time including the interprocedural edges that represent calls made from S , and ignoring those that represent calls made to S .

Extending that algorithm for interprocedural slicing to be consistent with the approach presented here is quite straightforward:

- The control-dependence edges in the PDGs should be computed using Definition 3.
- When the summary edges are computed via intraprocedural slicing, the slices should not follow control-dependence edges whose targets are pseudo-predicates.
- Similarly, when an interprocedural slice is computed, the control-dependence edges whose targets are pseudo-predicates should be followed only if the target is the source of the slice.

8 Experimental Results

To evaluate our work, we implemented Algorithms 2 and 3, and used each of them to compute slices in four C programs (information about the programs, the number of slices

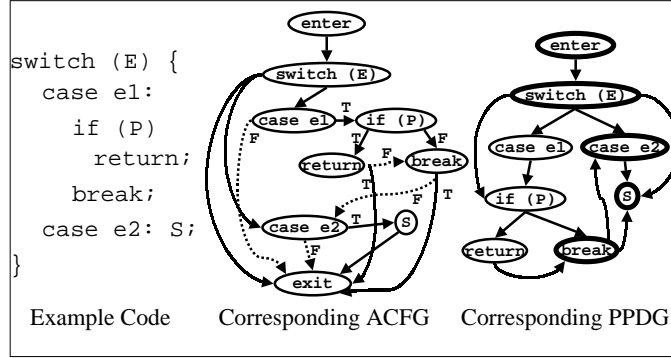


Figure 9: Example PPDG and its slice from S using Algorithm 3.

taken in each, and the average sizes of those slices is given in the table in Figure 10). Slices were taken from all of the nodes that could be reached by following one control-dependence edge forward from a node representing a switch case, and then following five data-dependence edges forward. This ensured that every slice would include a switch, but (by starting further along the chain of data dependences) avoided, for example, slices that would include *only* switch cases and breaks.

More details about the experimental results are given in the tables in Figures 11 and 12. Figure 11 presents information about the differences in the sizes of the individual slices taken using the two algorithms. The first column gives the number of cases where the two algorithms produced slices of exactly the same size. The other columns give the number of cases where the slice produced by Algorithm 2 was larger than the slice produced by Algorithm 3; the second column gives the number of cases where the size difference was between 1 and 10, the third column gives the number of cases where the size difference was between 11 and 20, etc.

Figure 12 presents information about how much the use of Algorithm 3 reduced the sizes of the slices. The first column gives the number of cases where there was no reduction in slice size (a 0% reduction). The other columns give

the number of cases where the reduction in size falls within the range specified by the previous and current column headers. For example, the second column gives the number of cases where there was a size reduction greater than 0% and less than or equal to 5%; the third column gives the number of cases where there was a size reduction greater than 5% and less than or equal to 10%.

Note that in almost all cases Algorithm 3 did produce smaller slices than Algorithm 2. Although this led to only a small reduction in the total size of the slice in most cases, there were some cases in both gcc.cpp and byacc where Algorithm 3 provided reductions in slice sizes of more than 15%, and some cases in flex where it provided reductions in slice sizes of more than 30%.

9 Related Work

Choi-Ferrante: The paper by Choi and Ferrante [3] that presents Algorithm 2 also includes a second algorithm: Given a node S , it starts with the slice from S computed using Algorithm 1, then adds `goto` statements to the slice to form a program that will always produce the same sequence of values for the variables used at S as the original program. This technique may produce smaller slices than those produced using Algorithm 2. However,

	lines of source code	number of APDG/PPDG nodes	number of slices	Av. slice size (# of nodes)	
				Alg 2	Alg 3
gcc.cpp	4,079	16,784	1,932	11,693	11,670
byacc	6,626	21,239	468	2,119	2,110
CADP	12,930	35,965	499	7,921	7,905
flex	16,236	31,354	1,716	8,150	8,082

Figure 10: Information about the C programs used in the experiments.

	0	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90
gcc.cpp	2	0	48	1881	0	1	0	0	0	0
byacc	0	229	239	0	0	0	0	0	0	0
CADP	18	152	160	169	0	0	0	0	0	0
flex	0	0	5	127	48	41	8	79	1405	3

Figure 11: Differences in slice sizes using the two algorithms.

	0%	5%	10%	15%	20%	25%	30%	35%
gcc.cpp	2	1918	3	1	8	0	0	0
byacc	0	438	18	7	5	0	0	0
CADP	18	481	0	0	0	0	0	0
flex	0	1572	0	5	13	52	66	8

Figure 12: Percent reduction in slice sizes achieved using Algorithm 3.

the `gotos` that are added are not necessarily in the original program; therefore, it satisfies neither Weiser’s definition of a correct slice, nor Definition 0 from Section 3.

Agrawal: Agrawal [1] also gives an algorithm that involves adding jump statements to the slice computed using the standard PDG, but the statements that he adds *are* from the original program. He states that this algorithm produces the same results as Algorithm 2; however, no proof is provided.

Harman-Danicic: More recently, Harman and Danicic [5] have defined an extension to Agrawal’s algorithm that produces smaller slices by using a refined criterion for adding jump statements (from the original program) to the slice computed using Algorithm 1. When applied to programs without switches, it may or may not produce slices that satisfy Definition 0. This is because their algorithm includes some nondeterminism: when there are cycle-free paths from a predicate to its immediate-postdominator both via its *true* and its *false* branches, then the jump statements along either of the paths can be chosen to be in the slice.

Unfortunately, when applied to programs with switch statements, this algorithm can be as imprecise as Algorithm 2. For example, when used to slice the switch statement in the first example in Figure 6, it produces exactly the same slice as Algorithm 2.

Another disadvantage of this algorithm as compared to ours is that the worst-case time to compute a slice can be quadratic in the size of the CFG, while our algorithm is linear in the size of the computed slice.

Sinha-Harrold-Rothermel: In [10], Sinha, Harrold, and Rothermel discuss interprocedural slicing in the presence of arbitrary interprocedural control flow; e.g., statements (like `halt`, `setjmp-longjmp`) that prevent procedures from returning to their call sites. That issue is orthogonal to the one addressed here (better slicing of programs with jumps and switches); thus, the two approaches can be combined to handle programs with arbitrary interprocedural control flow as well as jumps and switches.

10 Summary

We have provided a new definition for a “correct” slice, a new definition for control dependences, and a new slicing algorithm. The algorithm has essentially the same complexity as previous algorithms that compute slices using program dependence graphs, and is more precise than previous algorithms when applied to programs with jumps and switch statements.

The motivation for this work was the observation that slices of code with switch statements computed using the approach to handling jumps proposed by [2, 3] (as implemented in the CodeSurfer [7] programming tool) often include many extra components, which is confusing to users of the tool. We expect that the new approach will have an important practical benefit (to users of slicing tools) as well as being an interesting theoretical advance.

References

- [1] H. Agrawal. On slicing programs with jump statements. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 302–312, June 1994.
- [2] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Lecture Notes in Computer Science*, volume 749, New York, NY, November 1993. Springer-Verlag.
- [3] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. on Programming Languages and Systems*, 16(4):1097–1113, July 1994.
- [4] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [5] M. Harman and S. Danicic. A new algorithm for slicing unstructured programs. *Jrnl. of Software Maintenance*, 10(6):415–441, Nov./Dec. 1998.
- [6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence

graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, January 1990.

- [7] <http://www.codesurfer.com>.
- [8] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 177–184, 1984.
- [9] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. on Software Engineering*, 16(9):965–979, September 1990.
- [10] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Int. Conf. on Software Eng.*, pages 432–441, May 1999.
- [11] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.

Practical Model Checking and Example Generation for Context-Free Processes

James Ezick*
Cornell University
4139 Upson Hall
Ithaca, NY 14853
(607) 255-4934
ezick@cs.cornell.edu

David W. Richardson†
GrammaTech, Inc.
317 N. Aurora St.
Ithaca, NY 14850
(607) 273-7340
dwr5@grammatech.com

Tim Teitelbaum†
GrammaTech, Inc.
317 N. Aurora St.
Ithaca, NY 14850
(607) 273-7340
tt@grammatech.com

15 May 2001

Abstract

Context-Free Process Systems (CFPSs) are a natural model for the control flow of imperative programs. In this paper we present a space efficient refinement to an algorithm of Burkart and Steffen for model checking CFPSs over the full modal mu-calculus. Our algorithm works by generating a solution environment containing compact but complete solution information for every state in the model. By generating more information than is strictly necessary to verify a formula we show how it is possible to explore the space of examples in the model. We have implemented our approach as a plug-in for CodeSurfer, a code exploration tool, and used it to check formulas over interprocedural control flow graphs for actual C programs. The result is an interactive tool that for CTL formulas runs in time linear in the size of the model and that can extract precise valid program paths demonstrating the validity or non-validity of any part of the formula. For any mu-calculus formula the algorithm can be tuned to use as little as three bits per model state of computation space on top of the space for the model itself, the solution environment, and some additional space linear in the size of the formula.

1 Introduction

The fundamental component of any interactive tool designed to help explain the complexities of a piece of code is the ability to give insightful answers to precise questions about the relationships that exist between the program's statements. Model checking, a proven practical tool for the automatic verification of complex systems [6], is an ideal framework for such a tool. In this paper we present an extension to CodeSurfer® [1], a GrammaTech product, in which questions about C programs are cast in the modal mu-calculus and answers are derived by means of model checking over a context-free process system (CFPS), in our case the interprocedural control flow graph of the program.

Our approach to model checking is an extension and space efficient refinement to the approach of Burkart, Steffen, and Knoop [4, 5, 13]. Theirs is a *higher-order* approach that proceeds in two phases. First, property transformers are computed, one for each state in the model. Intuitively the property transformers capture the effect of a procedure on subformulas of the query over all possible contexts. They are used to express the set of subformulas that are valid at a state in a procedure relative to the set of subformulas valid at the end of the procedure. Finally, the model checking problem is decided by applying the property transformer of the start state of the model to the set of subformulas that are valid at the terminal points of the model. Their approach works over equation block systems [7], equivalent in expressiveness to formulas in the modal mu-calculus [14].

*Supported by NSF grants EIA-9726388, ACI-9870687, EIA-9972853, and ACI-0085969.

†Funded by DARPA contracts F30602-00-C-0080: Dependence Graphs for Information Assurance of Systems (OASIS), and DAAH01-01-C-R129: Verification of Hierarchical Graph Structures (SBIR).

In this paper we show how the size of the property transformers can be reduced using a partially intensional representation. Further we demonstrate how solution information for each state derived from the property transformers can be used to generate semantically meaningful paths that illustrate the validity or non-validity of any part of the formula. The result is a potentially powerful and practical interactive tool that software engineers can use to understand programs.

The rest of this paper is organized as follows. Section 2 defines context-free process systems and section 3 introduces the equation block form of the modal mu-calculus our algorithm uses. Our model checking algorithm is presented in section 4. Section 5 demonstrates how the the output of our algorithm can be used to generate examples of a formula in the model. In section 6 we present some relevant implementation details as well as discuss some experiences using our tool. Finally, in section 7 we contrast our work with some other approaches and discuss directions for future work.

2 Context-Free Process Systems

Context-free process systems can be thought of as a generalization of interprocedural control flow graphs. These systems form the models over which our algorithm operates. In this paper we will use process system and control flow graph terminology interchangeably. The context-free process systems we present are essentially the systems of Burkart and Steffen, however we do away with the terminology of *state-classes* which only has relevance when considering an expansion of the system. We also couple actions to process labels on edges. This coupling eliminates the requirement that processes be guarded by an initial action.

The equivalent of the the control flow graph for a single procedure is the *procedure process graph*.

Definition 1 A **procedure process graph (PPG)** is a quintuple $G = \langle \Sigma_P, Trans, \rightarrow_P, \sigma_P^{start}, \sigma_P^{end} \rangle$, where:

- Σ_P is a finite set of **states** (or CFG nodes)
- $Trans =_{df} Act \cup \mathcal{N}$ is a set of **transformations** where Act is a set of **actions** and \mathcal{N} is a set of **names**
- $\rightarrow_P =_{df} \rightarrow_P^{Act} \cup \rightarrow_P^{\mathcal{N}}$ is the **transition relation** where $\rightarrow_P^{Act} \subseteq \Sigma_P \times Act \times \Sigma_P$ and $\rightarrow_P^{\mathcal{N}} \subseteq \Sigma_P \times \mathcal{N} \times Act \times \Sigma_P$
- $\sigma_P^{start} \in \Sigma_P$ is a distinguished element, the **start state**
- $\sigma_P^{end} \in \Sigma_P$ is a distinguished element, the **end state**

For the remainder of the paper we use $\sigma \xrightarrow{a}_P \tau$ to denote $\langle \sigma, a, \tau \rangle \in \rightarrow_P^{Act}$. Similarly we use $\sigma \xrightarrow{(N,a)}_P \tau$ for $\langle \sigma, N, a, \tau \rangle \in \rightarrow_P^{\mathcal{N}}$. In either case we refer to τ as an *a-derivative* of σ . The start state, σ_P^{start} , is distinguished by not being a derivative of any other state in the PPG. The end state, σ_P^{end} , is likewise distinguished by not having any derivatives in the PPG.

To continue the control flow graph analogy, given $\sigma \xrightarrow{(N,a)}_P \tau \in \rightarrow_P^{\mathcal{N}}$ we refer to σ as a *call-node* and to τ as a *return-node*. Elements of $\rightarrow_P^{\mathcal{N}}$ are referred to as call-edges.

The definition of a *context-free process system* follows naturally as a binding of processes to names over a process set.

Definition 2 A **context-free process system (CFPS)** is a quadruple $\langle \mathcal{N}, \mathcal{P}, Act, P_0 \rangle$, where

- $\mathcal{N} = \{N_0, \dots, N_n\}$ is a set of **names**
- $\mathcal{P} =_{df} \{\mathcal{N}_i = P_i \mid 0 \leq i \leq n\}$ is a finite set of **PPG definitions** where the P_i are finite PPGs with names in \mathcal{N} .
- $Act =_{df} \bigcup_{i=0}^n Act_{P_i}$ is a set of **actions**
- $P_0 \in \mathcal{P}$ is a distinguished PPG, the **main PPG**, whose name, N_0 , is not an element of any \mathcal{N}_{P_i} .

```
#include <stdio.h>
```

```

1.  int G;
2.  void main() {
3.      int p=0;
4.      scanf("%d",&p);
5.      if((p%2)==0)
6.          fact(p);
7.      report();
8.  }

9.  int fact(int n) {
10.     if(n!=1) {
11.         int t = fact(n-1);
12.         G = n*t;
13.     } else
14.         G = 1;
15.     return G;
16. }

17. void report() {
18.     printf("%d\n", G);
19. }

```

Figure 1: sample.c

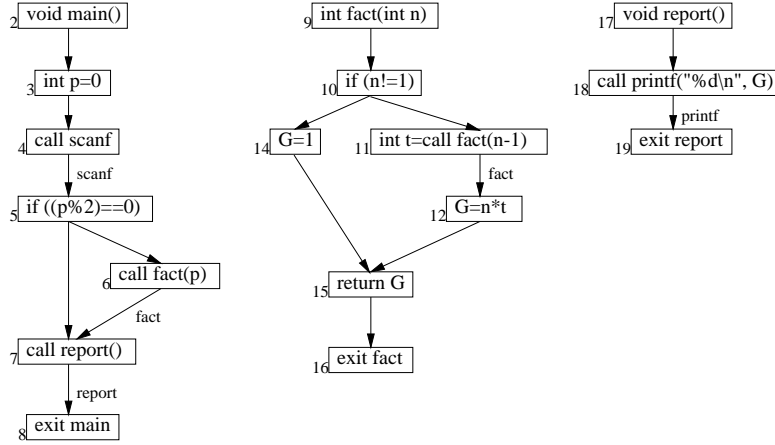


Figure 2: The context-free process system for sample.c

For the remainder of this paper the union of all PPG states, transitions, and call-edges in a CFPS will be denoted by Σ , \rightarrow , and \rightarrow^N respectively. Figure 1 is a sample C program that computes and prints a factorial for positive even integer inputs, prints garbage for odd inputs, and diverges for non-positive inputs. Figure 2 is the corresponding context-free process system over a single implied action.

Semantically, a *valid path* in the model is a path in the complete expansion of the system. The complete expansion is obtained by recursively substituting for each call edge, $\sigma \xrightarrow{(N_i, a)} \tau$, the PPG associated with N_i . An a -transition is introduced from σ to $\sigma_{P_i}^{start}$ and τ is merged with $\sigma_{P_i}^{end}$. For recursive programs the resulting complete expansion is infinite. Notice that for all programs the set of valid paths in the CFPS representation of their interprocedural control flow graph corresponds exactly to the set of potential execution paths in the program.

Finally, we introduce a strictly technical condition that is useful when model checking formulas derived from logics such as CTL. Let Σ^{term} be the subset of Σ consisting of $\sigma_{P_0}^{end}$ and every element of Σ with no derivatives that is not $\sigma_{P_i}^{end}$ for any P_i . In the complete expansion we will consider every element of Σ^{term} to be a derivative of itself under every element of Act . This ensures that, when restricting to a single action as we do for control flow graphs, every valid path can be infinitely extended.

The goal of the algorithm is then to declare a formula valid over a CFPS if and only if it is valid over the complete expansion of the CFPS - a structure that may be interpreted as a traditional Kripke transition system rooted at $\sigma_{P_0}^{start}$ [16]. In this way we can check the validity of temporal logic formulas with respect to only valid paths without any change to the logics themselves.

3 Equation Block Form of the Modal Mu-Calculus

The logic we use for posing questions in our system is the modal mu-calculus [14]. However, for technical reasons, we recast formulas as block equational systems [7, 8] before model checking. The translation to equational system form is a syntax directed process that operates over the parse tree of mu-calculus formulas in positive normal form. The result is a system of equations, the RHS of each corresponding to a subformula of the original mu-calculus formula. A designated variable, z_0 , corresponds to the entire formula. The size of the equation system is always linear in the size of the formula. In this section we review the syntax and semantics of equational systems.

Definition 3 Given Var , a countable set of variables each capable of representing a set of states, AP , a set of atomic propositions that is closed under negation, and Act , a set of actions (or modalities) a , **simple basic formula** is an expression of the form

$$\Phi ::= p \mid X_1 \mid X_1 \vee X_2 \mid X_1 \wedge X_2 \mid \langle a \rangle X_1 \mid [a] X_1$$

where $p \in AP$, $X_i \in Var$ for $i \in \{1, 2\}$, and $a \in Act$.

The semantics of the basic formulas is taken directly from their mu-calculus equivalents with one exception. Variables are interpreted with respect to an external environment, Env , that maps each occurring element of Var to a subset of Σ , the states in a model.

Definition 4 An **equation block** has one of two types, $\min\{E\}$ or $\max\{E\}$, where E is a list of mutually (def-use) dependent equations,

$$\langle X_1 = \Phi_1, \dots, X_n = \Phi_n \rangle$$

each X_i is a distinct element of Var , and each Φ_i is a simple basic formula.

We define the *weight* of an equation block to be the number of modal-defined variables in the block and we say that a variable, X_i is *free* in the block (or in a set of blocks) if it not the RHS of any equation in the block.

Further, we say that a block is *trivial* if it has weight zero and that a set of blocks is *homogeneous* if every non-trivial block is of the same type; a block set is *heterogeneous* otherwise.

Definition 5 A *closed equational system* is a list of equational blocks

$$\mathcal{B} = \langle B_1, \dots, B_m \rangle$$

in which the LHS of each equation is unique and in which no variable occurs free in \mathcal{B} .

We say that an equational system is *well-formed* if there are no cyclic def-use dependencies among the variable and junction-defined equations. Any mu-calculus formula can be translated into a well-formed equational system. For the remainder to this paper we will deal only with well-formed systems.

Figure 3 gives a mu-calculus formula and its corresponding equational block form. The formula is the mu-calculus translation of the CTL formula $A[\neg Use(G) \mathcal{U} Def(G)]$ over a , the forward control flow modality. Intuitively the atomic proposition $Use(G)$ corresponds to the statements in which global program variable G is used and $Def(G)$ to the statements where G is defined. In English, the formula asserts that on all computation paths, G is not used before it is defined and that it is eventually defined.

The translation to equation block form in figure 3 is not unique. It is permissible to switch the type of any trivial block. Further, any equation block can be partitioned into two or more blocks of the same type without changing the semantics. Likewise, any two blocks of the same type may be merged into a single block of that type [15]. Note, however, that splitting and merging blocks may introduce mutual block dependencies that make the interpretation more difficult to derive.

While we refer the reader to previous work [7] for the full precise semantics of equational systems, the central idea is that the system is interpreted as an environment, Env , mapping each variable to a set of states. Each block is interpreted as the portion of that environment restricted to the variables defined in it. The block environments are constructed by taking a least or greatest fixpoint, depending on the block type, of the equations in the block using the current approximation of the system's environment for the block's free

$$\begin{array}{lll}
\min \{ & z_5 = \neg Use(G) & \} & \max \{ & z_6 = [a]z_1 & & \min \{ & z_7 = [a]z_2 & \\
\min \{ & z_3 = Def(G) & \} & & z_4 = z_5 \wedge z_6 & & z_2 = z_3 \vee z_7 & \\
& & & & z_1 = z_3 \vee z_4 & & z_0 = z_1 \wedge z_2 & \}
\end{array}$$

$$\nu X.(Def(G) \vee (\neg Use(G) \wedge [a]X)) \wedge \mu Y.(Def(G) \vee [a]Y)$$

Figure 3: Mu-Calculus to Equation Block Translation

variables. The interpretation of the system is then the fixpoint reached by iterating over the blocks. The original formula is then valid at $\sigma \Leftrightarrow \sigma \in Env(z_0)$.

That the environment for each block and for the equational system in general converges to such a fixpoint is a consequence of the Knaster-Tarski fixpoint theorem [17]. The technique for generating an environment for a single block over a CFPS is the subject of the next section.

4 Algorithm

Given a CFPS, G , and a well-formed equational system \mathcal{B} , our algorithm returns an environment, $Env_{(G,\mathcal{B})}$, binding to each variable of \mathcal{B} the subset of Σ_G at which the variable is valid. As a preprocessing step the equation blocks are partitioned by their def-use relation into strongly connected components (SCCs) that are then topologically sorted. The algorithm proceeds by processing the SCCs in order, adding the variables defined in each SCC to $Env_{(G,\mathcal{B})}$. This ordering ensures that free variables of any SCC will already be completely solved in $Env_{(G,\mathcal{B})}$ when the SCC is processed. In this way the iteration process is localized to the SCC. Note that for hierarchical equational systems, as in figure 3, each SCC will be homogeneous and can therefore be reduced to a single block.

The processing of an SCC proceeds by iterating over its component blocks to a fixpoint from an initial approximation that assigns Σ to each variable defined in a *max* block and \emptyset to each variable defined in a *min* block. The one subtlety of the iteration is that each time a block in a heterogeneous SCC is processed it is necessary to first reset the approximations of the variables defined in it to their initial approximations. For homogeneous SCCs this reset can be shown to be unnecessary [9].

Before presenting the technique for processing a single block we make an observation: if \mathcal{B} is well-formed then for any $\sigma \in \Sigma$ the validity of any variable in \mathcal{B} can be derived from the validity of the atomic and modal-defined variables at σ . This follows immediately from the definition of a well-formed equational system. We will make use of this observation by restricting our property transformers to the modal-defined variables of a block.

The processing of an individual block is a three step process that follows Burkart and Steffen's original work [4] with modifications to incorporate access to free variables (which they omitted) and our partially intensional representation of the property transformers that makes use of our previous observation.

Let $\mathcal{X} = \{X_1, \dots, X_r\}$ be the set of modal-defined variables in the block being processed, with $\{X_1 = \Phi_1, \dots, X_r = \Phi_r\}$ the set of modal equations in the block. Further, let

$$PT_{init} = \begin{cases} PT_{null} = \lambda M. \emptyset & \text{if the block is a } min \text{ block} \\ PT_{univ} = \lambda M. \mathcal{X} & \text{if the block is a } max \text{ block} \end{cases}$$

1. *Construction of Property Transformers* We associate with each state in the model a property transformer (function) from the set of modal-defined variables in the block to itself. Intuitively, if PT_σ is the property transformer associated with a state σ and M is the set of modal-defined variables in the block that are valid at the unique end node of the PPG containing σ then $PT_\sigma(M)$ is the set of modal-defined variables that are valid at σ . Initializing each PT_σ to PT_{init} we iterate via worklist to the least or greatest fixpoint of the equations:

$$PT_\sigma = \begin{cases} PT_{init} & \text{if } \sigma \in \Sigma^{term} \\ PT_{id} = \lambda M. M & \text{if } \sigma \text{ is an exit state for some PPG } P_i, i \neq 0 \\ \nabla_{j=1, \dots, n} \{ \Delta_{(\sigma \xrightarrow{a} \sigma_j)} \circ PT_{\sigma_j} \} & \text{over all } \sigma_j \text{ derivatives of } \sigma \\ \{\sigma_j \mid 1 \leq j \leq n\} & \text{derivatives of } \sigma \end{cases}$$

where

$$\Delta_{(\sigma \xrightarrow{a} \sigma_j)} = \begin{cases} \delta_{\xrightarrow{\sigma_{P_i}^{start}}} \circ PT_{\sigma_{P_i}^{start}} & \text{if } \sigma \xrightarrow{a} \sigma_j \text{ is a call-edge to } P_i \\ \delta_{\xrightarrow{\sigma_j}} & \text{otherwise} \end{cases}$$

Let $M \subseteq \mathcal{X}$ and $\delta_{\xrightarrow{\sigma_j}}(M) = M'$ then

$$X_i \in M' \text{ iff } \begin{cases} \Phi_i = \langle a \rangle X_j & \text{and } X_j \in Expand(M, \sigma_j) \\ \Phi_i = [a] X_j & \text{and } X_j \in Expand(M, \sigma_j) \\ \Phi_i = [b] X_j & \text{and } a \neq b \end{cases}$$

Where $Expand(M, \sigma_j)$ is the union of the free variables of the block that hold at σ_j in the current environment and the set of variables defined in the block that hold at σ_j under the assumptions M and the values of the atomic propositions at σ_j .

Finally, let $M \subseteq \mathcal{X}$ and $(\nabla_{j=1, \dots, k} \{F_j\})(M) = M'$ then

$$X_i \in M' \text{ iff } \begin{cases} \Phi_i = \langle a \rangle X' & \text{and there exists } j \in \{1 \dots k\} \text{ with } X' \in F_j(M) \\ \Phi_i = [a] X' & \text{and } X' \in F_j(M) \text{ for all } j \in \{1 \dots k\} \end{cases}$$

The ∇ operation over a set of functions F_i can be thought of as a *mixed confluence operation* that returns a function equal to the meet of the F_i restricted to the \Box -defined variables and to the join of the F_i restricted to the \Diamond -defined variables.

2. *Solving for End and Return Nodes* Since the property transformers give the set of modal-defined variables valid at a state in terms of the modal-defined variables valid at the end state of the enclosing PPG, it is necessary to determine the set of modal-defined variables valid at the end state of each PPG. For recursive programs this requires an additional fixpoint iteration over the set of end and return states in the model. Using the obvious block-type dependent initial approximation again, we determine solutions for each return and end state according to the following equations:

$$M_{\sigma_{P_0}^{end}} = PT_{init}$$

If $M_{\sigma_{P_k}^{end}} = M', k \neq 0$ and $X_i = \Phi_i$ for all $i \in \{1, \dots, r\}$ then

$$X_i \in M' \text{ iff } \begin{cases} \Phi_i = \langle a \rangle X_j \text{ and } X_j \in M_{\sigma_j} \text{ for some } \sigma_j \text{ a return node of a call to } P_k \text{ in some PPG.} \\ \Phi_i = [a] X_j \text{ and } X_j \in \sigma_j \text{ for all } \sigma_j \text{ return nodes of calls to } P_k \text{ in some PPG.} \end{cases}$$

For each $\sigma^{return} \notin \Sigma^{term}$

$$M_{\sigma^{return}} = PT_{\sigma^{return}}(M_{\sigma_{P_j}^{end}}) \text{ where } P_j \text{ is the PPG containing } \sigma^{return}$$

3. *Solving for the Remaining States*

For each state σ in PPG P_i , $M_\sigma = PT_\sigma(M_{P_i}^{end})$. Since PT_σ was derived in step 1 and $M_{P_i}^{end}$ was derived in step 2 this is a straightforward function application.

In practice, since the property transformers depend upon the free variables of the block and since these are likely to change substantially with each iteration over an SCC, we discard the PTs after each iteration and retain only the results of step 3, which become part of $Env_{(G, \mathcal{B})}$.

The correctness of this algorithm follows from the well-formedness of \mathcal{B} and the proof of Burkart and Steffen [4]. While the complexity is exponential in both the maximal block weight and the alternation depth, both of these factors can be controlled in practice. For CTL, where the formula can always be translated into a hierarchical system with maximal block weight one, the algorithm is linear. Techniques for controlling the general complexity are covered in section 6.

5 Example Generation

In this section we demonstrate how the environment produced by our algorithm can be used to generate semantically meaningful examples that justify the answer to a model checking query. For this section we consider an environment, $Env_{(G, \mathcal{B})}$ to be the valid subset of $\Sigma \times \mathcal{V}$ where Σ is the set of states in a CFPS G and \mathcal{V} is the set of variables defined in an equational system \mathcal{B} .

The goal of example generation is to derive partial or full valid paths in the model that illustrate why a particular variable, state pair is or is not in the environment. This is done by searching an implicit environment dependence graph.

Definition 6 *The environment dependence graph \mathcal{K}_{Env} for an environment $Env_{(G, \mathcal{B})}$ over Σ and \mathcal{V} is a directed graph whose nodes are the elements of $Env_{(G, \mathcal{B})}$ and for which there is a possibly labeled edge*

$$(\sigma, v_i) \rightarrow (\tau, v_j) \text{ iff } \left\{ \begin{array}{ll} v_i = v_j & \text{and } \sigma = \tau \\ v_i = y \wedge z \text{ or } y \vee z & \text{and } (v_j = y \text{ or } v_j = z) \text{ and } \sigma = \tau \\ v_i = [a]v_j \text{ or } \langle a \rangle v_j & \text{and } \sigma \xrightarrow{a} \tau \in \rightarrow_{P_i}^{Act} \text{ for some PPG } P_i \in G \\ & \text{or } \exists \omega \in \Sigma \text{ with } \sigma \xrightarrow{(P_i, a)} \omega \in \rightarrow^{\mathcal{N}} \text{ and } \tau = \sigma_{P_i}^{start} \text{ (labeled push } \omega) \\ & \text{or } \exists \omega \in \Sigma \text{ with } \omega \xrightarrow{(P_i, a)} \tau \in \rightarrow^{\mathcal{N}} \text{ and } \sigma = \sigma_{P_i}^{end} \text{ (labeled pop } \tau) \end{array} \right.$$

When generating examples we will be walking \mathcal{K}_{Env} and when generating counter-examples we will be walking the complementary graph \mathcal{K}_{Env^C} .

Definition 7 *We define a walk W in an environment dependence graph \mathcal{K} as a sequence of pairs $\langle (\sigma, v), S \rangle$ where (σ, v) is a vertex in \mathcal{K} and S is a stack containing edge labels. Intuitively, a walk in \mathcal{K} is a path in which the edge labels act on the stack in the expected way and in which a “pop σ ” labeled edge can be traversed only if σ is the current top of stack or if the current stack is empty, denoted \perp . The pop action is defined to have no effect on an empty stack.*

A *terminal walk* W^T is a finite walk $W = \langle (a_0, S_0), \dots, (a_n, S_n) \rangle$ in which either there are no valid transitions out of (a_n, S_n) or in which (a_n, S_n) forms a loop in W . A *loop* is formed in W when there exists an $i \in \{0, \dots, n\}$ such that $a_i = a_n$ and S_i is a prefix of S_n . Loops capture the notion of infinite looping or recursion in a program. It is easy to show that there are no infinite non-terminating walks in \mathcal{K} .

Examples can then be defined as a subset of the terminal walks of \mathcal{K} .

Definition 8 *An example of (σ_0, v_0) in \mathcal{K} is a terminal walk $W^T = \langle ((\sigma_0, v_0), S_0 = \{\perp\}), \dots, ((\sigma_n, v_n), S_n) \rangle$ in which one of the following four conditions hold*

1. $\sigma_n = \sigma_{P_0}^{end}$
2. $((\sigma_n, v_n), S_n)$ forms a loop in W^T and either v_n is defined in a max block of \mathcal{B} and is valid at σ_n (example) or v_n is defined in a min block of \mathcal{B} and is not valid at σ_n (counter-example)
3. W^T is not a loop and either v_n is a \square -defined variable in \mathcal{B} and v_n is valid at σ_n (example) or v_n is a \diamond -defined variable in \mathcal{B} and v_n is not valid at σ_n (counter-example)
4. W^T is not a loop, v_n is an atomic-defined variable in \mathcal{B} , and either v_n is valid at σ_n and v_m is valid at σ_n under the assumption that every modal-defined variable \mathcal{B} is invalid at σ_n (example) or v_n is invalid at σ_n and v_m is invalid at σ_n under the assumption that every modal-defined variable in \mathcal{B} is valid at σ_n (counter-example) where m is the smallest m such that for all $i \in \{m \dots n\}$ $\sigma_i = \sigma_n$

Intuitively, the final condition checks whether the values of the atomic propositions at the final state are sufficient to cause the result. For valid CTL formulas over control flow graphs, examples correspond to execution paths whose infinite extension is valid with respect to a top-level path formula (a formula to the right of a top-level path quantifier).

An *example path* is the sequence of state transitions of an example. Figure 5 in appendix A illustrates what the user sees in response to the example query of figure 3.

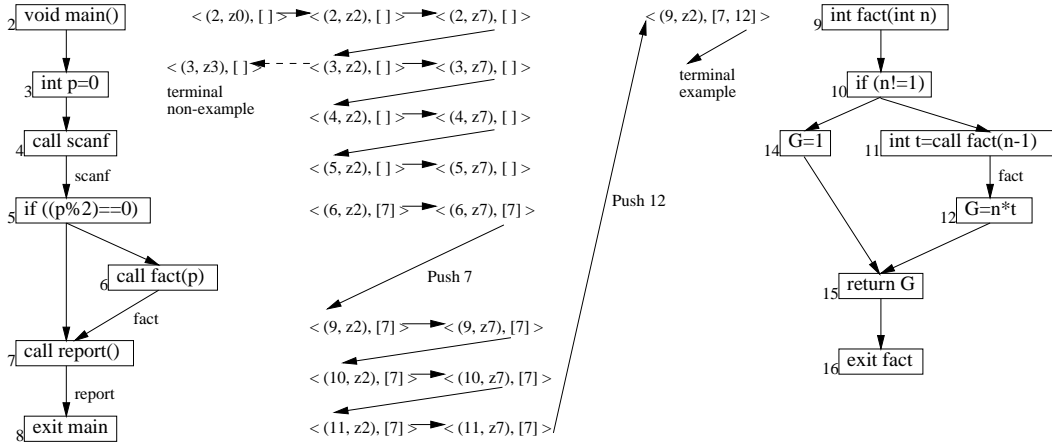


Figure 4: A Counter-Example in the Environment Dependence Graph

Figure 4 shows the counter-example path in the environment dependence graph for the example. The example satisfies condition 2 of our definition and can be reduced to an example path corresponding to an infinite recursive computation in which G is never defined. This path is realized by non-positive even inputs. Note that the terminal walk denoted by the dashed arrow in the figure is not an example since it fails to satisfy condition 4 of the definition - under the assumption that z_7 is valid at state 3, $z_2 = z_3 \vee z_7$ is valid at state 3 even though $z_3 = Def(G)$ is not.

6 Practical Considerations

Using CodeSurfer's built-in Scheme interpreter we have implemented a working prototype of both the model checking algorithm and the example generator. Our implementation works over interprocedural control flow graphs generated from C programs by CodeSurfer. As mentioned before, these graphs can be considered as CFPSs over a single action. These models are sufficient to handle the full C programming language provided we treat `longjmp` statements as simple function calls. Calls to library functions can either be bypassed or included in the model at the user's discretion.

Using these models and translations from high level temporal logics such as FCTL [10] we have produced a highly interactive tool in which a user can pose questions from an extensive predefined list of atomic propositions and then explore the space of examples using a graphical exploration tool built using the CodeSurfer API. In theory, atomic propositions can refer to any static property of program statements. This includes not only syntactic properties such as defined and used variables but also derived properties as well. For example, if the variable G in figure 1 had been passed to $fact(n)$ by reference rather than as a global variable we could have used alias information for $fact(n)$ to define nearly equivalent atomic propositions. CodeSurfer has built-in support for a wide range of such analyses.

In the remainder of this section we touch upon some of the implementation choices that we have made to improve the efficiency of our approach and discuss directions for future work.

6.1 The Model Checking Algorithm

As mentioned before, the model checking algorithm runs in time exponential in both the alternation depth of the formula and the maximal block weight of the system. The space required for the PTs is also exponential in the maximal block weight. In practice, however, alternation depths greater than two rarely occur. (CTL*, which contains both FCTL and LTL, is a strict subset of $L\mu_2$.) By using block splitting to trade off time for space the maximal block weight can be bounded by any constant the user defines. When this constant is set to one, the result is PTs that can each be explicitly represented in two bits. Adding one bit per state to denote inclusion in the worklist, the space overhead for the computation can grow at as little as three bits per state in the model.

For the environment we use a compact partially intensional representation. For atomic-defined variables the environment stores a closure representing the atomic proposition. Thus, for syntactic properties, the set of states associated with an atomic proposition is neither explicitly computed nor stored. For modal-defined variables, however, we are forced to store the actual sets. For every other variable type (junction and variable-defined) we store a closure that recursively computes its validity at a state by querying the validity of the RHS variables in the definition at the state. Thus, the set of states associated with these variables is represented completely intensionally. For CTL formulas over syntactic properties, such as in figure 3, the space required for the solution environment is one bit per state per modal-defined variable plus some additional space linear in the size of the formula for the intensionally represented variables.

6.2 The Example Generator

By default the example generator works by a modified depth first search in which we mark state, variable, stack triples instead of just vertices. This marking strategy allows us to explore through functions called from different contexts without backtracking. In theory, the number of example paths can be exponential in the size of the model. In practice, we generate a single example and then allow the user to redirect it from any intermediate point. In this way, the example can be “bent” toward any point of interest to the programmer.

Further, we use a hash table to store the state, variable, stack triples of the path currently being explored. When a new state, variable, stack triple is generated we can access the set of stacks associated with that state, variable pair in nearly constant time. We then only need to compare the elements with the new stack to check for loops. This eliminates a potentially cubic bottleneck in exploring a path.

6.3 Conclusions and Future Work

Our implementation is still in the prototype stages and presenting detailed performance data would be premature. Based on the success of the prototype, we are rewriting some of the algorithm’s key elements in C. Our knowledge of the performance that can be gained by vigorous optimization leads us to believe that this will be a practical tool for software analysis.

The primary weakness of our present approach is that control flow graphs specifically, and single-entry, single-exit PPGs in general, are inadequate for talking about the flow of data, particularly passed parameters, in a program. One of CodeSurfer’s original applications was program slicing based on system dependence graphs [12]. As such, CodeSurfer generates a wide range of data and control dependence information that we presently only utilize in trivial ways. This variety of relations between program statements was our motivation for choosing transition systems rather than state machines as our underlying model. One of the ways in which we plan to address this inadequacy is by extending our approach to CFPs based on multi-entry, multi-exit PPG’s. These generalize the system dependence graph in the same way that the current CFPs generalize the interprocedural control flow graph.

We feel that the strength of our approach comes from having available both the range of information CodeSurfer presently provides and the ability to generate a solution environment that distills that information.

7 Related Work

As previously mentioned, our approach to model checking is an extension of the work of Burkart, Steffen and Knoop [4, 5, 13]. Knoop showed how their approach could be modified so that only the PTs required to check a specific query are computed. Since our goal is to compute a complete set of solutions, this optimization has little application in our work.

Bebop [2], a product of the SLAM project at Microsoft Research, is another practical, and at this point more fully developed, approach to model checking C programs. Their approach operates over a Boolean program abstraction that captures control dependence as well as data dependence information we currently do not utilize. The abstraction they use is equivalent in power to a push-down automata.

VeriSoft [11], a product of Bell Laboratories and Lucent Technologies, is a verification tool for semantically exploring the state space of systems composed of concurrent processes executing C code. It operates via a state-less search using partial-order methods. At present we make no effort to deal with concurrent programs.

Unrestricted hierarchical state machines (UHSMs) [18, 3] are an alternate model for capturing the notion of context. They are similar to CFPs but without the labeling of transitions in the system. While these models are more amenable to automata [18] and reachability [3] based approaches to context-free model checking (which for some temporal logics are faster than our approach), the models themselves are less natural for representing the interplay between dependency types in a program. Specifically, they cannot readily be used to check mu-calculus formulas over multiple modalities, such as those derived from PDL.

References

- [1] www.grammatech.com/products/codesurfer/codesurfer_index.html.
- [2] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop (Lecture Notes in Computer Science No. 1885)*, pages 113–130, Zurich, Switzerland, September 2000. Springer-Verlag.
- [3] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. of ICALP 2001, Twenty-Eighth Int. Colloq. on Automata, Languages, and Programming (to appear)*, Crete, Greece, July 2001.
- [4] O. Burkart and B. Steffen. Model checking for context-free processes. In *International Conference on Concurrency Theory*, pages 123–137, 1992.
- [5] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221(1-2):251–270, 1999.
- [6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages (POPL)*, pages 117–126, 1983.
- [7] R. Cleaveland and B. Steffen. Computing behavioral relations, logically. *ICALP '91, LNCS 510*, 1991.
- [8] R. Cleaveland and B. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *CAV '91, LNCS 575*, pages 48–58, 1992.
- [9] E. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proc. 1st (IEEE) Symp. on Logic in Computer Science*, pages 267–278, 1986.
- [10] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, Louisiana, 1985.
- [11] P. Godefroid. Model checking for programming languages using VeriSoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.
- [12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
- [13] J. Knoop. Demand-driven model checking for context-free processes. In *ASIAN '99, LNCS 1742*, pages 201–213, 1999.
- [14] D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science (TCS)*, 27:333–354, 1983.
- [15] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, 1996.
- [16] M. Muller-Olm, D. Schmidt, and B. Steffen. Model checking: A tutorial introduction, 1999.
- [17] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- [18] M. Yannakakis and R. Alur. Model checking of hierarchical state machines. In *Proc. 6th ACM Symp. on Foundations of Software Engineering*, 1998.

A CodeSurfer GUI Screenshot

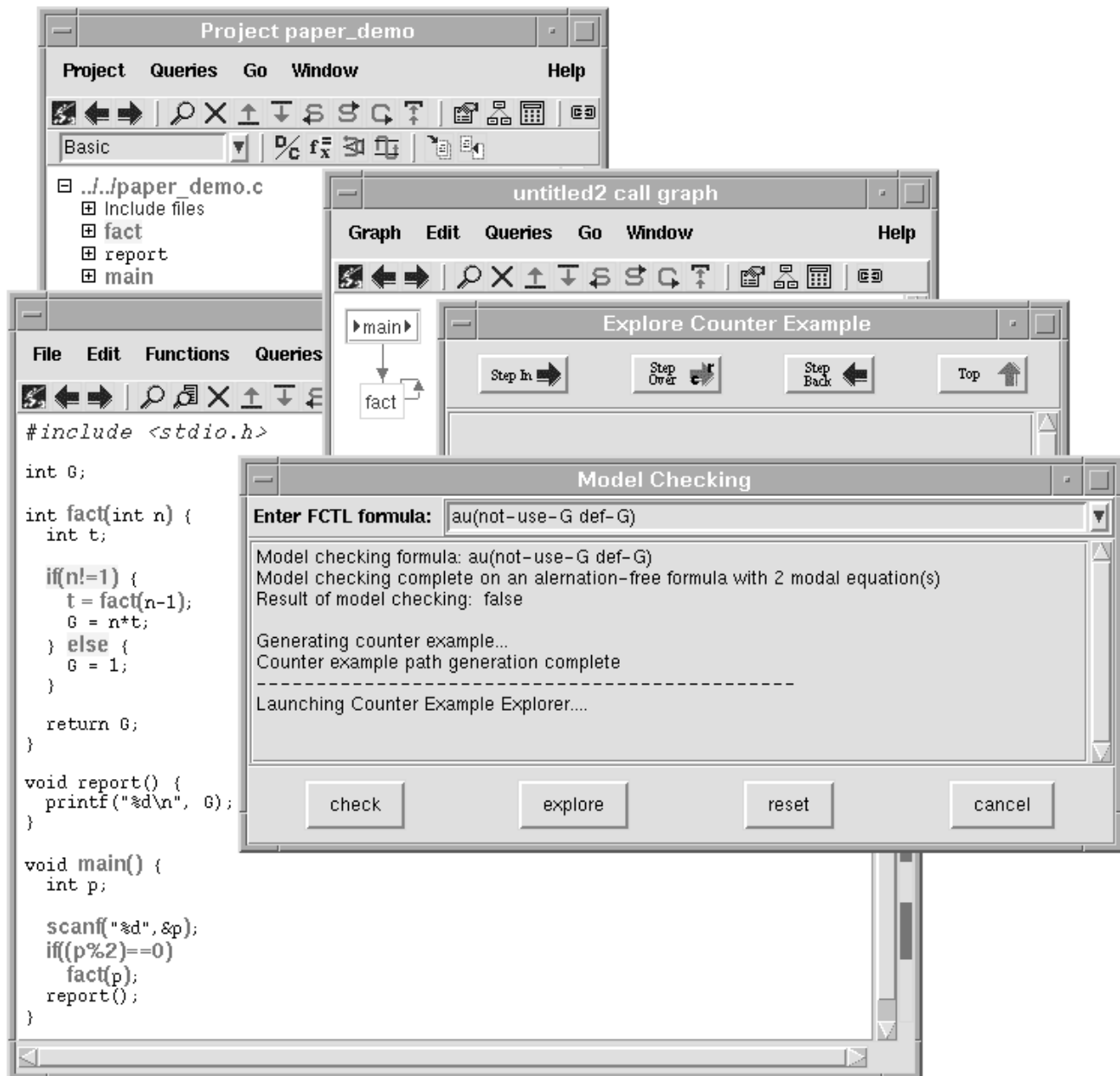


Figure 5: A screenshot of the model checker GUI illustrating the counter-example path of figure 4.

Software Inspection Using CodeSurfer

Paul Anderson, Tim Teitelbaum

I. INTRODUCTION

Software Inspection is a technique for detecting flaws in software before deployment. It was introduced by Fagan in 1976 [12], and since then its use has spread widely.

Despite the widespread adoption and success of software inspection, many software products continue to be released with large numbers of flaws. This can partly be attributed to the inherent complexity of software systems. The complexity of the software thwarts manual attempts to comprehend it.

Furthermore, the ideal situation for conducting software inspections in the field may often not be feasible. Time, geographical, or other constraints may mean that the original author of the code is not available to explain the structure of the code or describe its intended properties. Documentation may be misleading or even missing. General-purpose program understanding tools are crucial if code is to be inspected efficiently. However such tools until now have mostly operated on the surface-level syntactic features of the code.

Yet another difficulty is raised by the fact that safety or security requirements of software may be extremely difficult to show using manual techniques. For example, regulatory authorities that specify standards for safety-critical programs such as the Federal Aviation Authority (FAA) or the Nuclear Regulatory Commission (NRC) sometimes require that programs involved in the control of components have specific properties such as “part A must be independent of part B”. It is a difficult and error-prone process for a human to determine whether these properties hold for a program.

We believe that tools that allow reasoning about the deep structure of the code at a high level of detail will be extremely useful for doing software inspections. In this paper we describe how our own system—*CodeSurfer*¹—provides access to and queries on the system-dependence graph representation of a program for the purposes of helping with software inspections.

GrammaTech, Inc., 317 N. Aurora St., Ithaca NY 14850.
{paul,tt}@grammatech.com

Partially supported by DARPA contracts F30602-00-0080: Dependence Graphs for Information Assurance of Systems (OASIS), DAAH01-99-C-R192: Multi-Lingual Dependence-Graph Components for Software and Hardware Analysis, Design, and Specialization, and DAAH01-01-C-R129: Verification of Hierarchical Graph Structures (SBIR)

¹CodeSurfer is a registered trademark of GrammaTech, Inc.

The remainder of the paper is structured as follows. Section II presents some basic material on dependence graphs. Section III describes CodeSurfer—our system for program understanding. Section IV describes how queries on the system dependence graph can be used for software inspection. Section V describes using model checking techniques on the control-flow graph to reveal underlying flaws in the software. Section VI describes the ways in which CodeSurfer has been designed to be open and extensible. Section VII shows how this work relates to other work in software inspection. Finally, Section VIII concludes with a brief description of future work planned.

II. DEPENDENCE GRAPHS

Dependence graphs have applications in a wide range of activities, including parallelization [4], optimization [13], reverse engineering, program testing [2], and software assurance [17].

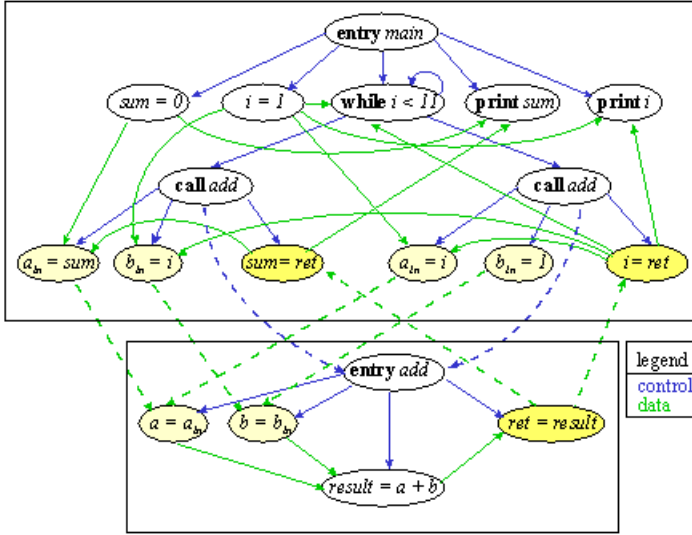
Figure 1 shows the dependence graph representation for a simple program with two procedures.

A *Program Dependence Graph* (PDG) [13] is a directed graph for a single procedure in a program. The vertices of the graph represent constructs such as expressions, call sites, parameters, and predicates. The edges between the vertices indicate either a data dependence or a control dependence. The data dependence edges are essentially data flow edges. For example, in Figure 1, there is a data dependence between the point `i=1` and the point `while (i < 11)` indicating that the value of `i` flows between those two points.

A control dependence edge between a source vertex and a destination vertex indicates that the result of executing the source vertex controls whether or not the destination vertex is reached. For example, in Figure 1, there is a control-dependence edge between the vertex representing the point `while (i < 11)` and the call site to the function `add`.

A *System Dependence Graph* (SDG) is a directed graph consisting of interconnected PDGs [18], one per procedure in the program. Interprocedural control-dependence edges connect procedure call sites to the entry points of the called procedure. Interprocedural data-dependence edges represent the flow of data between actual parameters and formal parameters (and return values).

Non-local variables such as globals, file statics, and variables accessed indirectly through pointers are handled by



```

void main()
{
  int sum, i;
  sum = 0;
  i = 1;
  while (i < 11) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  print(sum);
  print(i);
}

int add(int a, int
b)
{
  return(a+b);
}

```

Fig. 1. The System Dependence Graph for the small program shown on the right. Each function is represented as a collection of program points (shown in ovals) connected by edges (shown as arrows). There are different kinds of program points, e.g., entry, call-site, etc.

modelling the program as if it used only local variables. Each non-local variable used in a function, either directly or indirectly, is treated as a “hidden” input parameter, and thus gives rise to additional program points. These serve as the function’s local working copy of the non-local variable.

The PDG/SDG-based representation subsumes the notion of call graphs and data flow graphs. Numerous additional intermediate program representations are generated in the course of constructing a high-precision SDG for a program. These include the following:

- The program’s Abstract Syntax Tree (AST), with symbol table and full type information.
- The Control-Flow Graphs (CFG) and post-dominance graph.
- The Points-to Graph. This is a directed graph with vertices corresponding to variables (and structure fields, array elements and procedures), and edges indicating what values can point to which locations [22]. The pointer analysis algorithms used are those of Andersen [1], Steensgaard [35] and Das [8].
- Variable def/use information. The set of all variables whose values are taken or modified at all points in the program.
- The Call Multi-Graph. This includes calls made indirectly through function pointers variables.
- PDGs in which references to non-local variables of a procedure are modeled by turning such variables into extra (hidden) parameters.

A. Dependence Graph Queries

A number of queries on the dependence graphs are defined. The *backward slice* from a program point P includes all points that may influence whether control reaches P , and all points that may influence the values of the variables used at P when control gets there. The *forward slice* from P includes all program points affected by the computation or conditional test at P [37].

A program *chop* between a set of source program points S and a set of target program points T reveals how S can affect the state of the program at T [30].

These query algorithms can not be implemented using simple graph reachability — they must only return results that correspond to feasible executions of the program. A path that enters a procedure through a call site can only exit the procedure by going back to the call site from whence it came. We refer to queries on the dependence graph as being *precise interprocedural* if they follow this regime.

Precise interprocedural queries are implemented in CodeSurfer using *context-free language reachability* [29].

In order to do context-free language reachability on a graph, the edges in the graph are labelled with symbols. A *valid path* is one where the labels on the edges spell out a sentence in a context-free grammar.

It is a simple matter to construct a context-free grammar that models the call-return paths that correspond to the valid execution of a program. Let each call site in the program be given a unique index ranging from 1 through

N .

Let each interprocedural edge leaving from call site i be labelled $(i$, and each interprocedural edge returning to call site i be labelled $)_i$. Let all other edges be labelled x .

The grammar that gives rise to a precise interprocedural path in the SDG is one where the parentheses are matched. The following grammar specifies paths that are completely balanced by calls and returns:

$$\begin{array}{lcl} \text{matched} & \rightarrow & \text{matched matched} \\ & | & (i \text{ matched })_i \quad 1 \leq i \leq N \\ & | & x \\ & | & \epsilon \end{array}$$

A grammar that can be used to compute a slice can be written as follows:

$$\begin{array}{lcl} \text{realizable} & \rightarrow & \text{matched realizable} \\ & | & (i \text{ realizable })_i \quad 1 \leq i \leq N \\ & | & \epsilon \end{array}$$

Note that the starting point for a slice can be in a procedure F called by a procedure G . The grammar given above allows the path to proceed into F without having to return back to G .

Context-free language reachability is $O(n^3)$ in the number of edges in the graph. However, a preprocessing step computes *summary edges* that summarize the transitive dependence at call sites. This step, although also $O(n^3)$ in the number of edges allows precise interprocedural queries to be computed later in linear time [19].

Note that unstructured inter-procedural control flow (such as that induced by throwing exceptions) can be modeled this way as described in [34].

III. CODESURFER

CodeSurfer is a static analysis tool designed to support advanced program understanding based on the dependence-graph representation of a program. CodeSurfer is thus named because it allows surfing of programs akin to surfing the world-wide web.

CodeSurfer computes all of the above intermediate forms, and the entire system dependence graph for a program in advance. The dependence-graph queries discussed above are all implemented as primitive operations on the graph. All CodeSurfer operations operate by accessing these data structures directly, or by invoking the built-in dependence graph queries.

A number of viewers allow the user to access this information in a user-friendly manner. These viewers are connected by hypertext links. Some of the viewers are described below.

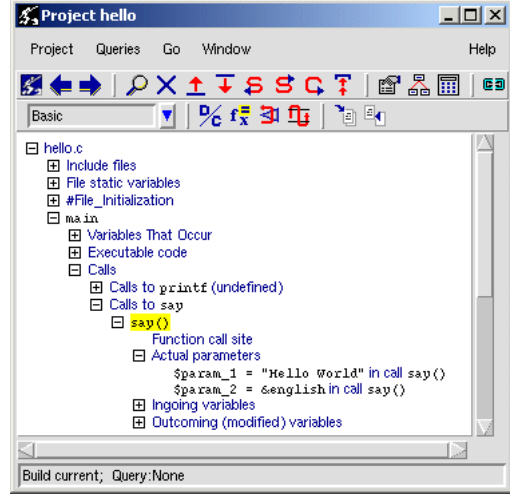


Fig. 2. A Project Viewer for a small program. This shows how CodeSurfer organizes the target code in terms of its program points.

- The *Project Viewer* shows the program organized hierarchically by file, then by function. Figure 2 shows a screen shot of the CodeSurfer project viewer.
- The *File Viewer* displays the source file. Tokens that give rise to vertices in the dependence graph are hypertext links in the file viewer. Figure 4 shows a file viewer for a small program.
- The *Call Graph Viewer* shows the call graph for the program. In this view, the edges are hyperlinks to all the call sites.
- *Property Sheets* are available for most program elements. For example, the property sheet for a variable will show where the variable occurs, where its value is used, where its value is assigned, where it may point to, and what other variables may point to it. Figure 3 shows a property sheet for a variable.
- The *Finder* allows searching through the program for occurrences of strings, or for particular functions or variables. For variables, the user can request declarations, occurrences, uses, and assignments. Attention can be restricted to globals, file statics, function statics, formal parameters, and/or locals. All variables that point to, or that are pointed to by, a given variable can be shown.
- The *Set Calculator* allows direct manipulation of the sets of points in the program. It provides a palette of logical set operations including as union, intersection and difference.

CodeSurfer provides a number of queries on the system dependence graphs that can be used for program understanding, or for finding flaws in the program. The next two sections describe these queries.

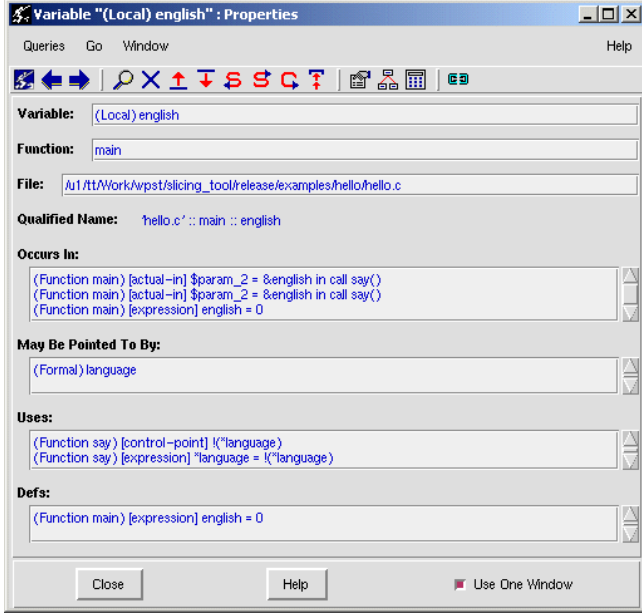


Fig. 3. A Property Sheet for a variable in the program. Most items shown are themselves hypertext links to other viewers. For example, selecting one of the Uses entries will navigate to a File Viewer and position it at that point.

IV. QUERIES FOR SOFTWARE INSPECTION

Many of the features of CodeSurfer have been designed for program understanding, and as such are useful for detailed software inspection. This section describes some of the queries and their application to software inspection.

A. Variable Usage Information

Each point in the program may access some variables or modify some variables, each possibly through pointers. In order to compute the data dependence graph, the set of variables used and defined at each program point are first computed and associated with the vertex that represents that program point.

This information is easily accessed by the user. For example, in Figure 3 shows the property sheet for the variable named **english**. The **Defs**: section of the property sheet indicates that the variable is only assigned to in one place—the expression **english = 0**. The **Uses**: part shows that the value of the variable has its value taken at two points, and that both of them are through a pointer.

Furthermore, the set of variables that can be used and/or modified for each procedure, either directly or transitively through a callee, is also computed. This can be used to answer questions of the form “Can global variable *G* be modified if function *F* is called”. The user can view this information directly through the Project Viewer.



Fig. 4. A File Viewer for a small program. In this example, the user has selected the first parameter to the first call to the procedure **say**, and has invoked a forward slice query. All points that depend on this parameter are shown in red

B. Predecessors/Successors

It is natural for a user attempting to understand a program to ask “How could variable *x* have gotten its value here?”, or alternatively “Where is the value generated at this point used?”. The predecessor and successor operations provide the answer to these questions. These queries can be posed for the control dependences, the data dependences, or both. A program point’s *data predecessors* are the points where the variables used at that point may have gotten their values. The *data successors* are the points where the variables that were modified at that point are used.

The predecessors and successors queries are implemented using the context-free language reachability algorithm on the dependence graph as described above in Section II-A.

The fact that the query is done directly on the dependence graph guarantees that the result will be correct with respect to the data flow properties of the program. For example consider the example in figure 5. If the predecessors query is invoked from line 5, lines 3 and 4 will be in the result. Line 1 will *not* be in the result because the value of *x* assigned on line 1 can never reach the use of *x* on line

5, because there is an assignment that kills it on line 3. Similarly the assignment to *w* on line 2 can never reach line 5 because of the kill on line 4.

```

1:   x = 100;
2:   w = x;
3:   x = 1;
4:   w = 10;
5:   z = x + w;

```

Fig. 5. A simple program fragment used to illustrate the built-in queries. The underlining indicates the starting point for the queries discussed in the text.

The fact that the the query is done using context-free language reachability guarantees that paths that do not correspond to valid paths through the program are not considered.

There are variants on these queries to narrow down the set of starting points for a query. *Point mode* is the default mode. As described above, a point mode query from line 5 yields lines 3 and 4.

Point-and-variable mode allows the user to restrict the set of starting points to those involving a set of variables. When invoked the user is prompted for the set of variables to be considered. For example, a point-and- variable mode predecessors query starting at line 5 in Figure 5 with respect to variable *x* yields line 3.

In *variable* mode the input is a set of variables and is independent of a starting point. In the example in Figure 5, a variable mode predecessors query with respect to variable *x* yields lines 1 and 2.

C. Slicing

A backward slice with respect to a set of starting points *S* answers the question “What points in the program does *S* depend on?”. The control dependence edges are used to determine how control could have reached *S*, and the data edges are used to determine how the variables used at *S* were computed. A forward slice with respect to a set of starting points *S* answers the question “What points in the program depend on *S*”.

Like the predecessor and successor operations, the slice operations have point and variable modes.

Slices are best used with care. Our experience is that the slicing operations are generally not used much for program understanding as they often to deliver too much information to be easily comprehended.

Figure 4 shows a CodeSurfer File Viewer where a forward slice has been invoked.

D. Chopping

A chop is a point-to-point reachability query in the graph. It answers the question “How does execution of

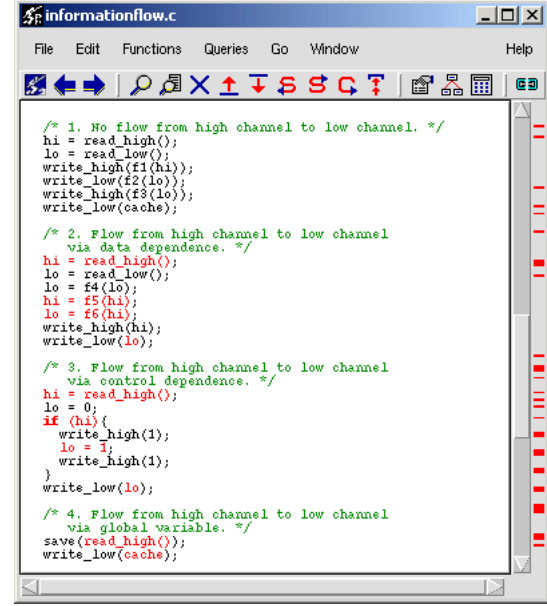


Fig. 6. The result of doing a chop shows how information can flow from one set of points to another. In this example, several points light up as being places where a security policy is being violated.

the program points at *A* affect the execution of *B*?”. This query can be used to determine the information flow between points in the program, or to show that two parts of the program are independent.

As mentioned previously, some regulatory agencies have software requirements that specify independence of components. In cases where the components are in the same program, a chop can be used to determine whether the software satisfies these requirements.

For example, the NRL network pump is a device that connects a high security network to a low security network [20]. The security requirement is that data can be transferred from the low side to the high side, but that no information can be allowed to flow from high to low. The exception is that communication acknowledgements *are* allowed to flow from high to low.

This property can be tested using the chop operation. The *sources* of the chop operation will be program points where data is read from the port connected to the high-security network. The *targets* of the chop operation will be the points where the data is written to the low channel. If the chop query returns the empty set, then this shows that the security property holds. If not, then the result is the set of points through which the property is violated.

Figure 6 shows the result of doing a chop on a mockup of the NRL pump. The points in red show those places involved in a violation of the security policy. There is no

flow in the first example. The second example is a blatant violation. It is useful to point out that the third example is more subtle. Fresh high data, which is read into `hi`, is used to change `lo` conditionally. The value of `lo` is then written to the low security port. Information is communicated from high to low via control dependence; low receives a 1 if and only if the high input was non-zero. Therefore a single bit of information has leaked. If high were 0-1 valued, this would be perfect information.

D.1 Red/Black separation

One form of independence property common in security applications is known as red/black separation. Values stored in a set of private variables (the red set) must not be allowed to flow to any of the set of public variables (the black set). This is a subtly different requirement than that expressed for the NRL pump. Here, the requirement is expressed in terms of the program's *variables* as opposed to *program points*.

CodeSurfer's *variable* mode can be used to help explore red/black separation properties. A variable-mode chop between a source set of variables V_S and a target set of variables V_T shows all ways in which information can flow between variables. This can be used to determine if a program conforms to the desired red/black separation policies.

V. MODEL CHECKING

Model checking is a technique widely used in digital hardware design to check properties of digital circuits [6]. We are adapting these techniques for doing model checking on programs with the goal of discovering programming flaws in systems. A full technical description of the model checker is beyond the scope of this paper. For a description, see [11]. Here we give a brief outline of the approach and describe how it can be used to answer questions that may be raised in detailed software inspections.

In model checking of digital circuits, model checkers operate on a graph where the vertices are the states of the circuit and edges represent state transitions. In contrast, the model-checking approach operates on the program's control-flow graph. Thus it can be thought of as checking the program in terms of all the possible paths through the program.

Unlike digital circuits, where the state space is "flat", the state space for a program's control-flow graph is constrained by the fact that when a call to a function finishes, control can only pass back to the point of call. This is the precisely the same issue that prevents straightforward graph reachability from being used to perform slicing queries, as described in Section II-A. In this case we use a space-efficient version of an algorithm due to Burkart and Steffen [3].

The model checking algorithm is capable of checking formulae in the full modal mu-calculus [21]. Translators can be written to convert formulae in higher-level logics such as Fair CTL into the mu-calculus.

The user interface to the model checker is neither of these logics, but instead a set of prepackaged or "canned" assertions about the behavior of the program in terms of valid execution paths, each of which is parameterisable. When invoked, the model checker evaluates the formula with its parameters, and if the assertion fails, the user is allowed to browse a counter-example in terms of a path through the code.

The parameters to the assertions are atomic propositions that can be specified in terms of the vertex being visited. For example, one query is "There exists a path where X holds until Y ". When invoked, the user is prompted to specify the propositions X and Y . These can be specified in terms of a set of predefined functions, or in terms of an arbitrary Scheme function. This function of course has access to the full system dependence graph, so sophisticated queries can be specified.

This mechanism can be put to use for posing queries useful in software inspection. One question that might be asked about a security-sensitive program is whether it contains a backdoor security vulnerability. If the application is a *login* program, then no user should be allowed access without having first gone through an authentication process. First the user would identify the points where the user is given access. These will typically be calls to a function. In the login program they might be calls to `exec()`. Let this set of points be named X . The user would then identify the points at which the authentication of the user is confirmed. Let these points be called Y . The canned query "No path goes through X without going through Y ". The resulting query will thus be "No path goes through (all calls to `exec()`) without going through (a call to `authenticate()`)".

Figure 7 shows a screen shot of CodeSurfer with the model checking interface being used to find possible sources of errors in a program.

VI. OPENNESS AND EXTENSIBILITY

CodeSurfer has been designed to be open and extensible where possible in order to foster users who wish to integrate with other tools, and to encourage users to build tools as add-ons to the system. The following sections describe the various ways in which CodeSurfer can be enhanced or extended.

A. Language.

The language-specific front-end to CodeSurfer produces an intermediate form that consists of a control-flow graph

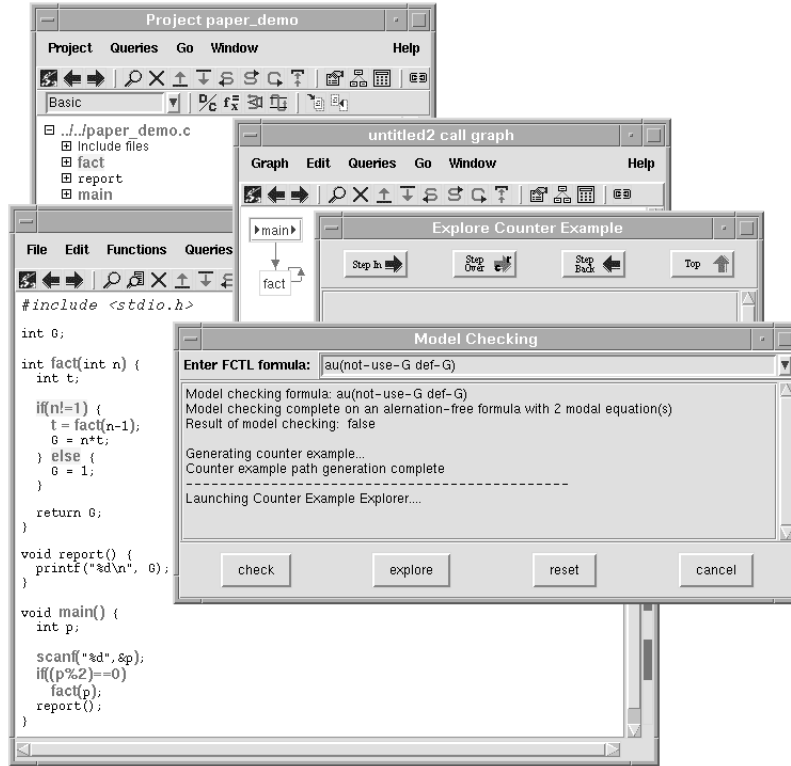


Fig. 7. The model checker has been used to make an assertion about the program which has failed. The GUI provides a means for browsing a path in the program counter-example. The interface for browsing the path is similar to a debugging tool being used to step through the execution of a program

annotated with variable usage information. As the front end is a separate executable, users can replace it with a front end for another language as long as programs in that language can be expressed in terms of the control-flow graph intermediate form. This has been done by CodeSurfer users for Jovial, Verilog, a subset of VHDL, and Promela—the language for the SPIN model checker [25].

B. Pointer Analysis.

CodeSurfer provides a choice of several pointer analysis algorithms, each of which offers different choices for precision and scalability. The pre-packaged algorithms include those of Andersen [1], Steensgaard [35] and Das [8]. As new pointer analysis algorithms are constantly being developed, the tool was designed so that new algorithms could be easily incorporated. As with the language-specific front end, the pointer analysis algorithms are packaged as separate executables, so they can be easily replaced by a user if necessary.

C. Scripting Language.

The CodeSurfer executable itself is written as a Scheme interpreter extended with the ability to create and manipulate system dependence graphs. The scheme interpreter is based on the STk implementation from Erick Galliesio at the University of Nice [15]. STk is fully integrated with the Tk widget set—the entire CodeSurfer GUI is written in Scheme using these widgets.

D. API to the Dependence Graph

The extensions to the Scheme interpreter introduce several new primitive types and provide a range of operations on them. These correspond to the underlying dependence graph data structures. For example, one type is the PDG; the operation (`pdg-vertices` *G*) returns the set of vertices associated with the program dependence graph *G*. Thus users can extend the system with new kinds of queries, or even new GUI elements. The model-checking application described in section V above is just such an extension.

E. Import/Export formats

CodeSurfer provides the ability to export arbitrary sets of program points to files in a range of different file formats. Additionally, some facilities are available for importing file formats and converting them to sets of program points. This mechanism facilitates integration with other tools. The set of standard formats is currently small, but growing. It currently includes *grep* format and *Pure-Coverage* format. A GXL filter [16] is planned for the future. This feature is also open and extensible. The user can define (in Scheme) new functions for converting sets of program points to external file formats and back again.

F. Set Calculator Operations

The set calculator provides the ability to manipulate sets of program points. The operations in the set calculator can be extended, again using Scheme, by an end user.

VII. RELATIONSHIP WITH OTHER WORK

Many Software Inspection Tools focus on groupware for the management of the software inspection process. These tools include ICICLE [33], ASSIST [23], Suite [9]. A comparison of such tools can be found in [24]. Few tools have been for detailed fine-grain inspection of software, although ICICLE does allow users to run *lint* on the C source files during the inspection.

Dunsmore [10] argues for a greater role of comprehension in the software inspection. There are many tools solely for program understanding [31], [36], [28], but we believe there are none that bring so much static analysis information to the user.

CodeSurfer has some commonality with tools for reverse engineering. These include the DMS Software Reengineering Toolkit [32], Datrix at Bell Canada [5], and the Portable Bookshelf [14]. However, unlike these systems, CodeSurfer makes no attempt to recover architectural information—its analysis is limited to creating a fine-grain system dependence graph. CodeSurfer does not have a general purpose meta-query system in the sense of DMS. Instead it makes do with basic context-free language graph reachability queries that are customizable programmatically. CodeSurfer does not have the ability to transform the program in the style of DMS, or TXL [7].

Other tools that provide a similar level of static analysis as CodeSurfer include other software re-engineering tools such as Refine [27] and Discover [26].

VIII. CONCLUSION AND FUTURE WORK

We have described a tool for inspecting and manipulating the dependence-graph representation of a program for the purposes of program understanding. We propose that

such a tool will be of use for doing formal software inspections. We have described the means by which the system answers queries about the data flow properties of the program using context-free language graph reachability. We have described using a model checker to answer questions about possible paths through the program.

Work on CodeSurfer is continuing under several research contracts. There are two main thrusts in the development of CodeSurfer. The first is to improve the scalability of the system. This will be achieved partly by improving the efficiency of the pointer analysis algorithms without sacrificing precision, and partly by using demand-driven techniques to reduce the up-front cost of building the dependence graph. The other thrust is to extend the domain of applications for the system. We are currently studying applying the technology to software assurance, and to program testing problems.

REFERENCES

- [1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).
- [2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Symp. on Princ. of Prog. Lang.*, pages 384–396, 1993.
- [3] Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. In *International Conference on Concurrency Theory*, pages 123–137, 1992.
- [4] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, 1986.
- [5] Bell Canada. <http://www.iro.umontreal.ca/labs/gelo/datrix>.
- [6] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
- [7] J. Cordy, I. Carmichael, and R. Halliday. The txl programming language, 1995 1995.
- [8] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PDLI'00*, Vancouver, BC, 2000.
- [9] J. Drake, V. Mashayekhi, J. Riedl, and W. Tsai. A distributed collaborative software inspection tool: Design, prototype, and early trial. Technical Report TR-91-30, University of Minnesota, August 1991.
- [10] A. Dunsmore. Comprehension and visualisation of object-oriented code for inspections. Technical Report EFoCS-33-98, Computer Science Department, University of Strathclyde, 1998.
- [11] James Ezick, David W. Richardson, and Tim Teitelbaum. Practical model checking and example generation for context-free processes. Submitted to the Workshop on Software Model Checking, Paris, France, July 23 2001.
- [12] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [13] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.
- [14] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.
- [15] Erick Gallezio. STk Home Page. <http://kaolin.unice.fr/STk/>.
- [16] Richard C. Holt and Andreas Winter. A Short Introduction to the GXL Software Exchange Format. In *WCRE 2000: Working Conference on Reverse Engineering*, Brisbane, Australia, Nov 6 2000.

- [17] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411. ACM, New York, May 1992.
- [18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
- [19] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, New York, NY, December 1994. ACM Press. Available at “<http://www.cs.wisc.edu/wpis/papers/fse94.ps>”.
- [20] M.H. Kang, I.S. Moskowitz, and D.C. Lee. A network pump. Technical report, Naval Research Laboratory, 1997. <http://www.itd.nrl.navy.mil/ITD/5540/-publications/CHACS/1997/1997kang-ACSAC97.ps>.
- [21] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [22] W. Landi, B. Ryder, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. Technical Report DCS-TR-336, Rutgers University, May 1998.
- [23] F. Macdonald. *Computer-Supported Software Inspection*. PhD thesis, Dept. Computer Science. University of Strathclyde, 1998.
- [24] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. In *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, Toronto, Canada, July 1995.
- [25] Lynn Millett and Tim Teitelbaum. Slicing promela and its applications to model checking, simulation, and protocol understanding. In *SPIN workshop.*, 1998.
- [26] MKS. MKS Home Page. <http://www.mks.com>.
- [27] Reasoning, Inc. Reasoning home page. <http://www.reasoning.com>.
- [28] Red Hat Software. The Source-Navigator IDE. <http://sources.redhat.com/sourcenav/>.
- [29] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November 1998. Special issue on program slicing.
- [30] T. Reps and G. Rosay. Precise interprocedural chopping. *SIGSOFT 95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Washington, D-C, October 10-13, 1995), *ACM SIGSOFT Software Engineering Notes*, 20(4), 1995.
- [31] Scientific Toolworks, Inc. Understand. <http://www.scitools.com/cpp.html>.
- [32] Inc. Semantic Designs. <http://www.semdesigns.com/Products/-DMS/DMSToolkit.html>.
- [33] V. Sembugamoorthy and L. Brothers. ICICLE: Intelligent code inspection in a c language environment. In *The 14th Annual Computer Software and Applications Conference*, pages 146–154, October 1990.
- [34] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *International Conference on Software Engineering*, pages 432–441, 1999.
- [35] B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, 1996.
- [36] Upspring Software. CodeRover Browser for C/C++. http://www.upspringsoftware.com/products/-coderover/browser_cpp.html.
- [37] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.